"Whoever called snooker 'chess with balls' was rude, but right"
– Clive James

University of Alberta

PICKPOCKET: AN ARTIFICIAL INTELLIGENCE FOR COMPUTER BILLIARDS

by

Michael Smith

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2006

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-22373-4
Our file Notre référence
ISBN: 978-0-494-22373-4

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

Billiards is a game of both strategy and physical skill. To succeed, a player must be able to select strong shots, and then execute them accurately and consistently on the table. Several robotic billiards players have recently been developed. These systems address the task of executing shots on a physical table, but so far have incorporated little strategic reasoning. They require artificial intelligence to select the 'best' shot taking into account the accuracy of the robotics, the noise inherent in the domain, the continuous nature of the search space, the difficulty of the shot, and the goal of maximizing the chances of winning. This thesis describes the program PickPocket, the winner of the simulated 8-ball tournaments at the 10th and 11th Computer Olympiad competitions. PickPocket is based on the traditional search framework, familiar from games such as chess, adapted to the continuous stochastic domain of billiards.

Acknowledgements

My many thanks go out to the following people who had a hand in this work:

Michael Greenspan, whose research with Deep Green brought about the opportunity to undertake this project. Without his organizing of both the software infrastructure upon which this work was built, and the computational 8-ball competitions in both Taiwan and Italy, all that follows would not have been possible.

Jonathan Schaeffer, my supervisor, for all the support, encouragement, and feedback throughout the course of the project.

Will Leckie, Marc Godard, and Jean-Francois Landry - 'the competitors', for the friendly competition and good company in Italy! Special thanks to Will for his work behind the scenes maintaining the *poolfiz* library.

Fredrik Niemela, for his assistance running the tournaments in both Taiwan and Italy.

The members of the GAMES group at the University of Alberta, many of whom have shown enthusiasm for the project and provided suggestions which have found their way into this work.

And finally, NSERC and iCore, which provided the funding to support this work.

Contents

1 Introduction 1
- 1.1 Billiards . . . 1

2 Background 4
- 2.1 Robotic Billiards . . . 4
- 2.2 The Search Framework . . . 5
  - 2.2.1 Traditional Search . . . 6
  - 2.2.2 Search in Stochastic Games . . . 7
- 2.3 Billiards Physics Simulation . . . 9
- 2.4 8-ball . . . 10

3 PickPocket Implementation 13
- 3.1 Move Generation . . . 13
  - 3.1.1 Straight-in Shots . . . 17
  - 3.1.2 Bank Shots . . . 18
  - 3.1.3 Kick Shots . . . 18
  - 3.1.4 Combination Shots . . . 21
  - 3.1.5 Shot Difficulty . . . 21
  - 3.1.6 Velocity Table . . . 25
  - 3.1.7 Safety Shots . . . 26
  - 3.1.8 Bank/Kick/Combo Activation . . . 27
  - 3.1.9 Root Sampling . . . 27
- 3.2 Evaluation Function . . . 28
- 3.3 Search Algorithms . . . 30
  - 3.3.1 Probabilistic Search . . . 30
  - 3.3.2 Monte-Carlo Search . . . 33
  - 3.3.3 Search Enhancements . . . 35
- 3.4 Game Situations . . . 37
  - 3.4.1 Break Shot . . . 37
  - 3.4.2 Ball-in-Hand . . . 39

4 Experimental Results 42
- 4.1 Error Model . . . 42
- 4.2 Sample Size and Statistical Significance . . . 43
- 4.3 Experiments . . . 45

4.3.1 Probability Table 47
4.3.2 Evaluation Function 49
4.3.3 Safety Shots 51
4.3.4 Root Sampling 52
4.3.5 Probabilistic Search 53
4.3.6 Monte-Carlo Search 53
4.3.7 8.5-ball 56
4.3.8 Bank, Kick, Combination shots 59
4.3.9 Search Enhancements 60
4.3.10 Search Algorithm Comparison 61
4.4 Computer Olympiad 10 63
4.5 Computer Olympiad 11 64

5 Conclusions and Future Work **68**
5.1 Future Directions for Work 68
5.1.1 Adaptation to Other Billiards Games 68
5.1.2 Implementation Enhancements 71
5.1.3 Man-Machine Challenge 74
5.2 Conclusions 76

Bibliography **77**

List of Tables

4.1 Confidence Intervals for 100-Game Match Result 45
4.2 Probability table accuracy 48
4.3 Probability table match result 49
4.4 2-ply comparison of evaluation functions 50
4.5 1-ply comparison of evaluation functions 51
4.6 Safety match result 51
4.7 Root sampling match result 52
4.8 Effect of search depth in probabilistic Search 54
4.9 Effect of sample size in 1-ply Monte-Carlo search 55
4.10 Effect of sample size in 2-ply Monte-Carlo search 55
4.11 Effect of search depth in Monte-Carlo search 55
4.12 Effect of search depth in Probabilistic 8.5-ball 58
4.13 Effect of search depth in Monte-Carlo 8.5-ball 58
4.14 Bank/Kick/Combo match result 59
4.15 Effect of pruning in Probabilistic search 60
4.16 Effect of pruning in Monte-Carlo search 60
4.17 Comparison of search algorithms 62
4.18 Computer Olympiad 10 competition results 64
4.19 Computer Olympiad 11 competition results 65

List of Figures

2.1 Generic search algorithm 6
2.2 A standard billiards table 11

3.1 Parameters defining a billiards shot 14
3.2 A straight-in shot 15
3.3 A bank shot 15
3.4 A kick shot 15
3.5 A combination shot 16
3.6 Shot difficulty parameters 22
3.7 Effect of β on effective pocket size 23
3.8 Rail interactions with corner and side pockets 23
3.9 Probabilistic search algorithm 32
3.10 Monte-Carlo search algorithm 34
3.11 Probabilistic search w/ pruning 36
3.12 Monte-Carlo search w/ pruning 38
3.13 Ball-in-hand algorithm 41

4.1 Error vs. Sample Size for 50-50 Win Rate 44

Chapter 1

Introduction

1.1 Billiards

Billiards refers to a family of games played on a billiards table. Players use a cue stick to strike the cue ball into an object ball, generally with the intent to drive the object ball into a pocket. The most popular billiards games are 8-ball pool, 9-ball pool, and snooker. There are a wide variety of other games that can be played with the same equipment, including straight pool, one-pocket, and cutthroat.

Billiards games emphasize both strategy and physical skill. To succeed, a player must be able to select strong shots, and then execute them accurately and consistently. Several robotic players have recently been developed, including Deep Green [1] and Roboshark [2]. These systems address the task of executing shots on a physical table, but so far have incorporated little strategic reasoning. To compete beyond a basic level, they require AI to select the 'best' shot to play in any given game situation.

Three main factors determine the quality of a billiards shot. First, it must contribute towards the player's goals. Most shots sink an object ball, allowing the player to shoot again and progress towards clearing the table. Safety shots, giving up the turn but leaving the opponent with few viable shots, are another strategic option. The many potential extraneous shots that perform neither of these have little value. Second, the shot's difficulty is a factor. All else being equal, shots with a high probability of success are preferred. Finally, the quality of the resulting table state after the shot is a factor. A shot that leaves the player well positioned to make an easy follow-up shot on another object ball is preferred.

Skilled human billiards players make extensive use of position play. By consistently

choosing shots that leave them well positioned, they minimize the frequency at which they have to make more challenging shots. This makes it easier for them to pocket long consecutive sequences of balls. Strong players plan several shots ahead. The best players can frequently run the table off the break shot. The value of lookahead for humans suggests a search-based solution for building a billiards AI. Search has traditionally proven very effective for games such as chess. Like chess, billiards is a two-player, turn-based, perfect information game. Two properties of the billiards domain distinguish it, however, and make it an interesting challenge.

First, it has a continuous state and action space. A table state consists of the position of 15 object balls and the cue ball on a continuous <x,y> coordinate system. Thus, there are an infinite number of possible table states. This renders standard game-tree search enhancements inapplicable. Similarly, each shot is defined by five continuous parameters: the aiming direction, velocity, cue stick elevation angle, and x and y offsets of the cue stick impact position on the cue ball, so there are an infinite number of possible shots available in any given table state. A set of the most relevant of these must be selectively generated.

Second, it has a stochastic nature. For a given attempted shot in a given state, there are an infinite number of possible outcomes. The player can visualize their intended shot, but will always miss by a small and effectively random delta amount. A professional player, trained for accuracy, will tend to have small deltas, whereas casual players will exhibit larger deltas. Similarly, for a robotic player, deviations from the intended shot arise from limitations in the accuracy of the vision and control systems. Ambient environmental factors such as temperature and humidity can also affect collision dynamics, leading to variance in shot outcomes. This stochastic element means that a deterministic expansion of a move when building a search tree, as is done in chess, is insufficient to capture the range of possible outcomes.

This thesis describes PickPocket, an artificial intelligence program for computer billiards. PickPocket was created to compete in the simulated billiards tournaments held at the 10^{th} and 11^{th} Computer Olympiads. It won both of these competitions. The program is based on the traditional game search framework, and is an adaptation of these techniques to the continuous, stochastic domain of billiards. Specifically, this thesis presents:

- A shot generator for billiards,

- An evaluation function for billiards,
- Two search algorithms for billiards: one based on Expectimax and one based on Monte-Carlo search,
- Pruning optimizations for both search algorithms,
- A new approach to estimating the shot difficulty function,
- Experimental results confirming the benefit of search over the previously standard greedy approach, and
- Experimental results exploring the effect of PickPocket's many parameters and features.

Chapter 2 gives background details on robotic billiards, the search framework, the simulated billiards domain, and the rules of 8-ball. Chapter 3 describe's PickPocket's implementation, including its shot generator, evaluation function, search algorithms, and domain specific implementation details. Chapter 4 gives the results of a range of experiments with PickPocket, as well as details of the Computer Olympiad competition results. Chapter 5 concludes this thesis with a discussion of how PickPocket could be adapted to play other billiards games, and gives some directions for future work.

Chapter 2

Background

2.1 Robotic Billiards

Several recent robotics projects address the task of physically executing shots on a billiards table [1, 2, 3, 4]. These systems use machine vision to assess the table state, and robotic control of a cue stick to execute shots. A high degree of accuracy is required in both of these areas for competent play, making billiards an interesting and challenging domain for robotics research.

In these systems, the cue stick is generally controlled by an overhead mounted gantry robot. Roboshark uses a gantry with four degrees of freedom: x position, y position, z position (cue stick height), and aiming direction. Deep Green has five degrees of freedom; the elevation angle of the cue stick is also controlled. An end effector is used by both systems to drive the cue stick into the cue ball to make shots[1]. With this setup, the robot has as much control over the cue stick position as a human player does.

All robotics billiards players so far have used vision to assess the table state, primarily by means of an overhead mounted global vision camera. The images captured by this can be processed to identify the location of the balls on the table. Note that this can be a non-trivial task; in 8-ball, there are solid and striped object balls of the same colour. From certain angles, these can be hard to differentiate. Deep Green additionally features a local vision camera mounted along the cue stick's line-of-sight. This aids in fine-tuning the shot once the cue stick is almost in position.

The decision algorithms that these robotics projects have used for shot selection have

[1] Roboshark actually uses a pneumatic cylinder rather than a regulation cue stick to hit the cue ball, but the effect is the same

so far been relatively simplistic. They focus on finding the *easiest* shot, rather than the *best* shot. That is, they attempt to find the shot with the highest probability of success, without regard for position or strategic play. The algorithms described in [5, 6, 7, 8] are all greedy algorithms which generate a set of shots, assign a difficulty score to each shot, and select the easiest shot to execute. Their methods for assigning a difficulty score to each shot are described in Section 3.1.5.

Several programs have been developed concurrently with PickPocket to compete in the Computer Olympiad computational billiards tournaments. These are detailed in Section 4.5.

Two other related projects are Stochasticks [9], and Larsen's automatic pool trainer [10]. These are systems that aid humans in planning shots, and deal with vision problems like those faced by billiards robots.

2.2 The Search Framework

Any search-based game-playing program consists of three main components: a move generator, an evaluation function, and a search algorithm. From any given game state, these components work together to construct a *search tree* which selects a move to take. A search tree is a type of tree graph consisting of nodes which correspond to states in the game, and edges which correspond to moves, the transitions between states in the game. When searching, the move generator generates a set of legal moves for each non-terminal node. The evaluation function assigns a score corresponding to the value of that state for the player to each *leaf* node, the nodes at the depth being searched to. The search algorithm defines in what order nodes are expanded, and how each node propagates the values of its children up the search tree.

The initial game state being searched corresponds to the *root node* of the search tree. A *ply* of search refers to all of the nodes at a given *search depth*, or number of moves from the root node. For example, a 1-ply search consists of the root node and all of its children. These children are leaf nodes. A 2-ply search consists of the root node, all of its children (which are interior nodes), and all of its children's children (which are leaf nodes). The *branching factor* of a game tree refers to the average number of children (moves) at each

```
float Generic_Search(GameState state, int depth){

  // if a leaf node, evaluate and return
  if(depth == 0) return Evaluate(state);

  // else, generate moves
  moves[] = Move_Generator(state);

  GameState nextState;

  // search each generated move
  foreach(moves[i]){
    nextState = Apply_Move(moves[i], state);

    // save the score for this child node
    scores[i] = Generic_Search(nextState, depth - 1);
  }

  // return a function of the child scores
  return F(scores);
}
```

Figure 2.1: Generic search algorithm

node. Pseudocode for a generic search algorithm is shown in Figure 2.1. This implements a basic search with nodes expanded in depth-first order. The return value at a node is a function of the scores of its child nodes. Most typically this would be the maximum child score, but it can vary based on the search algorithm being implemented.

If at a leaf node, the game is a win or a loss, an absolute evaluation can be returned. Otherwise, a *heuristic* evaluation, an estimate of the value of the state for the player, must be returned. The more accurate this heuristic, the better the program will play.

2.2.1 Traditional Search

Search has proven very effective in games such as chess and checkers. The chess program Deep Blue famously defeated world champion grandmaster Garry Kasparov in a 1997 match [11]. The checkers program Chinook became world champion in 1994 [12]. These programs use highly tweaked and optimized variants of the basic *Minimax* search algorithm. Minimax is an algorithm for games where the turn alternates back and forth

between players after each move. Most traditional games have this *adversarial* property. *Max* nodes correspond to states where it is the searching player's turn. The player wants to maximize their score, so the value propagated up the tree is the maximum of the scores of the child nodes. *Min* nodes correspond to states where it is the opponent's turn. The opponent is also trying to win, so will choose the best move from their perspective. This is the worst option from the player's perspective, so the minimum child score is propagated upwards at min nodes.

Pruning search trees by not searching branches that are provably inferior to a branch already seen can provide a large performance boost. The $\alpha\beta$ algorithm implements pruning in Minimax search trees [13]. There are also a wealth of other optimizations that take advantage of the deterministic structure of Minimax-style trees.

2.2.2 Search in Stochastic Games

Stochastic games have in common the presence of random events that effect the play of the game. These random events can come in two forms: they may determine *what actions* a player has available to them (such as the roll of the dice in backgammon, or the drawing of tiles in Scrabble), or they may influence the *outcome* of the actions the player takes (such as in billiards shots, or the drawing of cards in poker). This non-determinism complicates search. Whenever a stochastic event occurs, instead of one successor state in the search tree, there are many. The search algorithm must take into account both the value to the player of each possibility, and the likelihood of it occuring. Two established approaches to dealing with this are Expectimax search and Monte-Carlo sampling.

Expectimax Search

Expectimax, and its *-Minimax pruning optimizations, are an adaptation of game tree search to stochastic domains [14] [15]. Expectimax search operates similar to standard Minimax search, with the addition of *chance nodes* wherever a non-deterministic action is to be taken. For example, dice rolls in the game of backgammon would be represented by chance nodes. Chance nodes have a child node for each possible outcome of the non-deterministic event. In backgammon, every possible outcome of the dice roll would have a corresponding child node. The value of a chance node is the weighted sum of its child

nodes, where each child is weighted by its probability of occuring. This accounts for both the value of the child and its probability of occuring.

Monte-Carlo Sampling

A Monte-Carlo sampling is a randomly determined set of instances over a range of possibilities. Each instance is assigned a value, and the average of all values is computed to provide an approximation of the value of the entire range. This implicitly captures likelihood of good and bad outcomes (as a state that has mostly strong successor states will score highly, and vice versa), as opposed to Expectimax which explicitly uses probabilities to factor in the likelihood of events occuring.

Monte-Carlo techniques have been applied to a wide range of problems. In game-playing, they have been used in card games such as bridge [16], poker [17] and hearts [18], as well as Go [19] and Scrabble [20]. All of these games have in common a branching factor too large for traditional search. All except Go have a stochastic element. To explicitly build full search trees would be impossible. In card games, the number of possible deals of the cards is massive. Rather than accounting for all of them explicitly, a set of instances of deals are sampled. In Go, the number of moves available to the player is too large to search deeply, so random games are played out to estimate the values of moves. In Scrabble, possible tile holdings are sampled from the tiles known to be left in the bag, rather than searching every possibility. The Scrabble program Maven [20] also uses *selective sampling*, using a biased rather than uniform sampling to improve play.

Monte-Carlo search has also been investigated for continuous real-time strategy (RTS) games [21]. Unlike the turn-based games mentioned so far, these are real-time games where players take their actions simultaneously. This increases the complexity of the domain dramatically. In turned-based games, players have the benefit of having a static game state which they can spend time analyzing before taking an action. In contrast, the environment in an RTS is constantly changing. Turn-based games naturally impose a rigid structure on search trees. RTS actions can occur in any sequence, or simultaneously, so knowing how to structure a search tree is a difficult problem. RTS games feature a near-infinite branching factor to compound the challenge.

Stochastic games where search has previously been investigated fall therefore into

two categories: turn-based games with discrete state and action spaces like card games, backgammon, and scrabble; and continuous real-time RTS games. Unlike the former, billiards has a continuous state and action space. Unlike the latter, billiards has a rigid turn-based structure. Also unlike card games where an opponent's hand is typically hidden, billiards features perfect information. Therefore billiards bridges the complexity gap, bringing together elements of traditional deterministic perfect information games like chess, stochastic games like backgammon, and realtime continuous games. It is 'more complex' than backgammon because of its continuous nature, yet 'less complex' than RTS games because of its turn-based structure. It is the first game with this particular set of properties to be examined from an AI perspective. There is a family of such turn-based continuous stochastic games, which include croquet, lawn bowling, shuffleboard, and curling. The techniques and considerations discussed here for billiards should carry over to these domains.

2.3 Billiards Physics Simulation

The outcome of any billiards shot depends on the physical interactions between the balls moving and colliding on the table. The physics of billiards are quite complex, as the motion of balls and results of collisions depend on the spin of the ball(s) involved as well as their direction and velocity. Leckie's *poolfiz* [22] is a physics simulator that, given an initial table state and a shot to execute, finds the resulting table state after the shot completes.

Poolfiz implements an event-based physics simulation. In this approach, equations modeling billiards ball dynamics are solved to find the time of the next event. Events include ball-ball collisions, ball-rail collisions, and ball motion state transitions (such as from moving to stopped). Simulation time is then advanced to this point, the positions of the balls are located, the event is resolved, and then the equations are reapplied to find the time of the next collision. This is repeated until all balls are at rest and the shot is complete. This method is a departure from the numerical integration method that has been the traditional approach to this type of physics simulation.

Simulation results are deterministic, whereas the outcome of shots made by a human or robot player on a physical table are non-deterministic. To capture this stochastic element,

the input shot parameters to *poolfiz* are perturbed by a noise model at game time. This results in a slightly different shot outcome every time for a given set of input parameters. The noise model is described in Section 4.1.

Simulation is a costly operation; it is where PickPocket spends a vast majority of its time. To save on runtime simulation costs, PickPocket precomputes several tables for performing common tasks. These are the shot probability table, minimum velocity table, and rail rebound table. Each is covered in detail in Chapter 3.

2.4 8-ball

PickPocket plays 8-ball, as this was the game selected for the computational billiards competitions held at the 10^{th} and 11^{th} Computer Olympiad. While many of the implementation details that follow are specific to 8-ball, the overall approach could be easily applied to any billiards game. The adaptations to the search framework to account for billiards' stochastic and continous nature would be the same. Specific considerations for several other billiards variants are discussed in Section 5.1.1.

Billiards games are played on a rectangular table, twice as long as it is wide. Typical dimensions for a table used for 8-ball and 9-ball range from $3' \times 6'$ to $4.5' \times 9'$. Snooker is played on a larger table, typically measuring $6' \times 12'$ if full sized. A billiards table has six pockets; four in the corners and two midway along the sides. It is also marked by a headstring line, $1/4$ of the distance along the length of the table, and a footstring line $3/4$ of the length along the table. At the midpoint of the headstring and footstring are the head spot and the foot spot. Often the footstring is not physically depicted on a table, but implied by the location of the foot spot. Figure 2.2 depicts a standard billiards table.

PickPocket plays by the rules of 8-ball as standardized by the Billiards Congress of America (BCA) [23]. The major rules can be summarized as follows:

- Fifteen numbered object balls are initially racked in a triangular formation, with the 1-ball positioned on the foot spot and the triangle extending out behind the footstring.
- The 1-ball through 7-ball are solid colours, and collectively referred to as 'solids'. The 9-ball through 15-ball feature a coloured stripe on a white background, and are

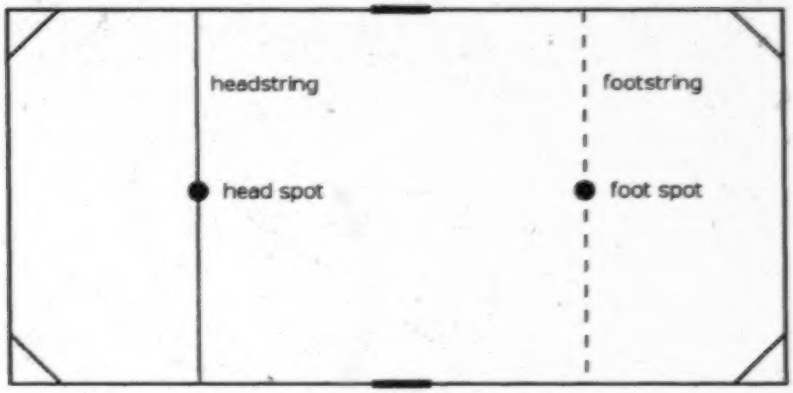

Figure 2.2: A standard billiards table

referred to as 'stripes'. Each player will be assigned one of these groups during the game, and to win must pocket all seven balls in their group, followed by the 8-ball.

- One player is selected to break, or shoot first. They shoot at the racked balls with the cue ball from behind the headstring. For an 8-ball break to be considered legal, it must either pocket a ball or at least 4 numbered balls must contact a rail. If the breaker pockets any object ball on the break, they continue shooting; otherwise, they lose their turn.

- Immediately after the break shot, the game is in an 'open table' state. Players are not yet assigned to solids or stripes, and may shoot at any ball on the table.

- 8-ball uses a called-shot rule. That is, the shooter must indicate which ball they intend to sink in which pocket. They continue shooting if they pocket the called object ball in the called pocket, otherwise they lose their turn.

- The first time a player calls and successfully pockets a stripe or solid, they are assigned to that group. Their opponent is assigned to the other group.

- For a shot to be legal, the cue ball must first contact a ball of the player's assigned group before one of the opponent's group. Additionally, a numbered ball must be pocketed or any ball must contact a rail after this first cue-object ball collision for the shot to be legal.

- An illegal shot is called a foul. The player that shot the foul loses their turn, and their opponent gets 'ball-in-hand', the right to place the cue ball anywhere on the table.
- If a player scratches, or pockets the cue ball, their opponent gets ball-in-hand.
- Pocketed balls stay pocketed, even if pocketed by a foul shot.
- A player may call 'safety' on any shot and force his opponent to take the next shot.
- Pocketing the 8-ball at any time before a player has cleared their group is an automatic loss. If a player has cleared their group and is shooting at the 8-ball, and scratches or fouls on a shot that pockets the 8-ball, they suffer an automatic loss.
- The first player to pocket all of their assigned group of balls, followed by legally pocketing the 8-ball, wins the game.

Chapter 3

PickPocket Implementation

Any search-based game-playing program consists of three main components: a move generator, an evaluation function, and a search algorithm. This chapter discusses the adaptation of each of these to a stochastic continuous domain, and in particular the implementation used by PickPocket. Two search algorithms are presented: Probabilistic search (a special case of Expectimax) and Monte-Carlo sampling search. These algorithms have offsetting strengths and weaknesses, representing the classic trade-off between breadth vs. depth in search.

3.1 Move Generation

A move generator provides, for a given game state, a set of moves for the search algorithm to consider. For deterministic games like chess, this is often as simple as enumerating all legal moves. For games with a continuous action space, it is impossible to enumerate all moves; a set of the most relevant ones must be selectively generated.

In deterministic games, an attempted move always succeeds. A chess player cannot 'miss' when capturing an opponent's piece. In billiards, shots vary in their difficulty. Shots range from ones players rarely miss, such as tapping in a ball in the jaws of a pocket, to very challenging, such as a long bank shot off a far rail. This difficulty is a key property of the shot itself, and thus must be captured by the move generator. With every shot generated, it must provide an assessment of its difficulty. This is used by both the evaluation function and the search algorithm to perform their respective tasks, as described in Section 3.2 and Section 3.3.

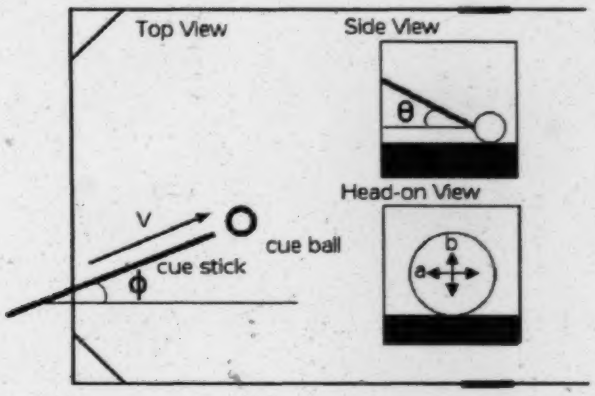

Figure 3.1: Parameters defining a billiards shot

The need to selectively generate relevant shots, and to assign a difficulty assessment to generated shots, arise respectively from the continuous and stochastic nature of the billiards domain.

Every billiards shot is defined by five continuous parameters, illustrated in Figure 3.1:

- ϕ, the aiming angle,
- V, the initial cue stick impact velocity,
- θ, the cue stick elevation angle, and
- a and b, the x and y offsets of the cue stick impact position from the cue ball centre.

Shots that accomplish the goal of sinking a given object ball into a given pocket can be divided into several classes. In order of increasing difficulty, they are: The straight-in shot (Figure 3.2), where the cue ball directly hits the object ball into the pocket; the bank shot (Figure 3.3), where the object ball is banked off a rail into the pocket; the kick shot (Figure 3.4), where the cue ball is banked off a rail before hitting the object ball into the pocket; and the combination shot (Figure 3.5), where the cue ball first hits a secondary object ball, which in turn hits the target object ball into the pocket. Theoretically these can be combined to arbitrary complexity to create multi-rail bank and combination shots. In practice, difficulty increases rapidly with each additional collision, so players only attempt the more challenging types of shots when they lack easier options.

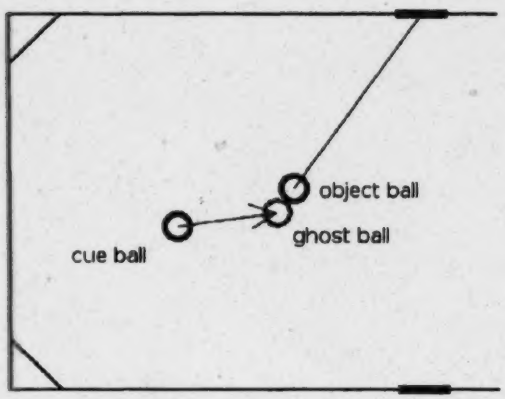

Figure 3.2: A straight-in shot

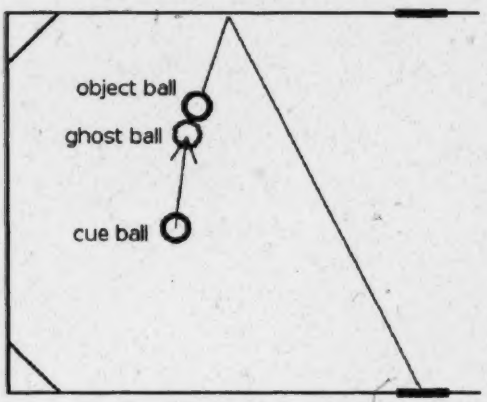

Figure 3.3: A bank shot

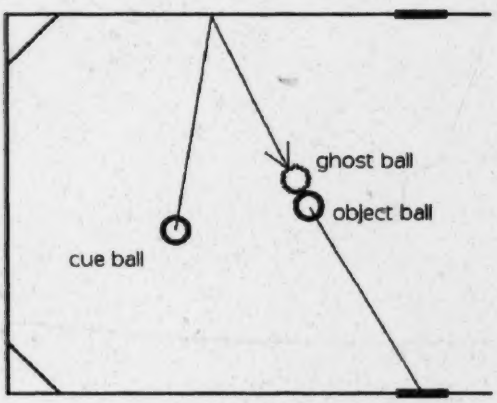

Figure 3.4: A kick shot

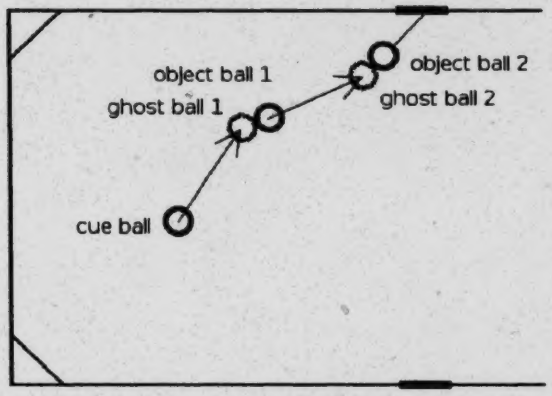

Figure 3.5: A combination shot

The target object ball is aimed with the aid of a conceptual 'ghost ball'. A line is extended from the intended target position (the center of a pocket, for example) through the center of the target object ball. The position adjacent to the object ball on the far side of this line from the target position is the ghost ball position. If the cue ball is aimed such that it is in this position when it contacts the target object ball, the target object ball will travel in the direction of the target position post-collision. Therefore, the ϕ aiming angle for the cue ball is chosen so that it is aimed directly at the ghost ball position to drive the object ball in the desired direction.

In addition to controlling the direction of the target object ball, the shooter has a significant degree of control over where the cue ball ends up after a shot. Consider a straight-in shot. As a result of billiard ball dynamics, ϕ largely determines the shape of the shot up until the cue ball's first collision with the target object ball. This object ball will have a similar post-collision trajectory regardless of the values of the other parameters. However, the cue ball's post-collision trajectory can be altered by varying V, a, and b, which affect the cue ball's spin at the time of collision. V at the same time affects the distance travelled by the cue and object balls. It must be sufficiently large to sink the desired object ball, while variations above this threshold determine how far the cue ball travels post-collision. θ is constrained by having to be large enough that the cue stick is not in collision with either any object balls on the table or the rails around the table's edge. High θ values can impart curvature on the cue ball's initial trajectory.

3.1.1 Straight-in Shots

PickPocket generates shots one class at a time, starting with straight-in shots. For every legal object ball, for every pocket, a straight-in shot sinking that object ball in that pocket is considered. Sometimes this shot is not physically possible. This can occur when the object ball is not between the cue ball and the target pocket, or when another object ball blocks the path to the pocket. If the cue ball is very near or frozen against[1] another object ball, it is impossible for the cue to hit it from some directions. Checks are made for these conditions, and impossible shots are discarded. When the shot is possible, the parameters are set as follows:

- ϕ is chosen such that the object ball is aimed at the exact centre of the pocket.
- V is retrieved from a precomputed table of minimum velocities necessary to get the object ball to the pocket. See section 3.1.6.
- θ is set to a minimum physically possible value, found by starting at 5° and increasing in 5° increments until the cue stick is not in collision with any other object balls or the side of the table. Most of the time this turns out to be 5° or 10°. θ has the least impact on shot trajectory, so relatively large 5° increments are used to quickly find a physically possible value.
- a and b are set to zero.

This generates exactly one shot sinking the target ball in the target pocket. An infinite set of these could be generated by varying the shot parameters, especially V, a, and b, such that the altered shot still sinks the target ball in the target pocket. Each variation on the shot leaves the table in a different follow-up state. For position play, it is important to generate a set of shots that captures the range of possible follow-up states. PickPocket discretely varies V, a, and b to generate additional shots. For example, V is increased in $1m/s$ increments up to *poolfiz's* maximum $4.5m/s$. The number of variations per ball and pocket combination has a strong impact on the branching factor when searching. These values were hand-tuned such that the 2-ply Monte-Carlo search algorithm from Section 3.3.2 would execute within the time limits imposed at the Computer Olympiad.

[1] In contact with.

If one or more straight-in shots are found, move generation is complete. If not, Pick-Pocket falls back on the other shot classes in order of increasing complexity until shots are found. A vast majority of the time, straight-in shots are found and this is not necessary. Section 4.3.8 details experiments which show that bank, kick, and combination shots totalled under 6% of the total shots played under typical conditions.

3.1.2 Bank Shots

Bank shots are generated in a manner similar to that described for straight-in shots. Instead of aiming the target object ball directly at the pocket, it is aimed at a point along the rail such that it will reflect into the target pocket. For an object ball-rail collision, the angle of incidence is approximately equal to the angle of reflection - this simplification is used to solve for the target position along the rail.

For every legal object ball, for every pocket, for every rail, a bank shot is considered sinking the object ball into the target pocket off the selected rail. It is physically impossible to bank off a rail into a pocket along that same rail, so one rail is skipped for side pockets, and two rails are skipped for corner pockets. Additionally, the object ball must be between the cue ball and the target rail for the shot to be physically possible.

Since a minimum velocity table for bank shots would be too large to be reasonable, the velocity is initially set to $2m/s$ for these shots. This is sufficient for the vast majority of bank shots. As with straight-in shots, it is then incremented, creating new shot variants up to $4.5m/s$ velocity.

3.1.3 Kick Shots

Kick shots are significantly more complicated to calculate than bank shots. Unlike bank shots, the assumption that the angle of incidence is approximately equal to the angle of reflection for the cue ball-rail collision does not hold strongly. The exact angle of reflection depends on the amount of spin on the cue ball at the time of collision. This is a function of the initial velocity V at which the cue ball is struck, and the distance between cue ball and the rail. The angle of reflection may vary by up to several degrees from the angle of incidence. Therefore a simple geometric calculation is not sufficient to calculate an effective kick shot.

The amount of spin on the cue ball at collision time varies because of the dynamics of a billiards shot. Immediately after the cue ball is struck by the cue, it slides along the table felt without rolling. This state is called 'stun'. Gradually friction between the cue ball and table felt causes the cue ball to start rolling along the table. In this transitional phase, its forward motion is partially sliding motion, and partially rolling motion. Eventually the cue ball transitions into a state where its forward motion is entirely rolling motion, called 'normal roll'. The angle of reflection off the rail will vary depending on the point along this transition that the cue ball is in at the time of collision, as each point has different spin characteristics. The shooter can also use side spin to influence the angle of reflection.

To generate a kick shot, a ghost ball position which aims the target object ball at the desired pocket can be found by the same method used for straight-in shots. Then an aiming point along the rail (which determines ϕ) and an initial velocity V for the shot must be found, such that the centre of the ghost ball falls on the line extending from the cue ball-rail contact point in the direction of the angle of reflection. Holding either ϕ or V fixed, varying the other influences the angle of reflection. The shot is also constrained by having to be of sufficient velocity to sink the target object ball. For a given cue ball-rail contact point, the velocity that results in the correct angle of reflection may be too weak to pocket the target object ball. Because of the interaction between ϕ and V, there may be many possible kick shots that sink a given object ball in a given pocket. PickPocket generates one such shot, with the goal of finding a kick shot that is robust.

The angle of reflection for a shot at a rail with an initial ϕ and V cannot be determined geometrically; it must be found through simulation. Rather than perform expensive simulations at runtime to optimize ϕ and V for a given kick shot, PickPocket precomputes a rail rebound table and retrieves values from this table to find kick shot parameters. To generate the table, the angle of incidence, cue ball-rail distance, and initial velocity parameters are discretized and a shot is simulated for every physically possible combination of these three parameters. The angle of reflection and cue ball velocity immediately after reflection are recorded. Table entries are indexed by their angle of incidence, angle of reflection, and cue ball-rail distance. In each table entry, the minimum and maximum initial velocity V to achieve that angle of reflection given the angle of incidence and cue ball-rail distance are recorded, as well as the minimum velocity the cue ball can have after the collision

with the rail. During precomputation, this table is updated after each combination of initial parameters is simulated.

At runtime, kick shots are generated with the aid of lookups into this table. A set of potential cue ball-rail contact points are found, equally spaced and centred around the point where the angle of incidence (the cue ball-rail angle) equals the angle of reflection (the ghost ball-rail angle). For each of these points, one lookup into the table is made, for the specific angle of incidence to that point, required angle of reflection from that point to the ghost ball position, and distance between the cue ball's initial position and that point. If an entry exists, the minimum rebound velocity of the cue ball from the rail is checked to ensure it is sufficient. This is made by making a lookup into the minimum velocity table as though the shot were a straight-in shot with the cue ball initially located at the rail contact point; if the minimum rebound velocity is greater than the minimum velocity required for the shot, then this rail contact point is stored as a good candidate. If multiple good candidate contact points are found, the one with the largest range between the minimum velocity and maximum velocity to get the desired reflection angle is selected. The larger this range, the more robust the shot - the less sensitive it is to small changes in the initial parameters. The velocity in the center of the minimum-maximum range of the selected good candidate is selected as V for the shot, as this is the value with the largest margin of error on both sides. If at least one good candidate entry in the table is found, this method finds a unique ϕ and V for a kick shot. If no such entries are found, a kick shot is not generated.

When generating kick shots, a and b are not varied - each of these affects the spin of the cue ball when it impacts the rail, and hence its angle of reflection. To generate kick shots by varying these parameters, PickPocket would have to optimize for four parameters concurrently instead of two. θ is set, as usual, to a minimum physically possible value.

Because the cue ball's angle of reflection off the rail is sensitive to small changes in the initial shot parameters, the rail rebound table must be calculated to a very fine granularity. PickPocket uses a $900 \times 900 \times 100$ table, giving an accuracy to 0.1 degrees for the angles of incidence and reflection, and 1cm for the cue ball-rail distance. The table is stored using a sparse data structure to save memory - the vast majority of table entries are empty. For a given angle of incidence, only those angles of reflection within several degrees are likely to be populated.

This process results in kick shots being generated by a sequence of table lookups, rather than the alternative of a sequence of runtime simulations. When generating kick shots, a shot is considered for every combination of target object ball, target pocket, and kick rail. A set of the physically possible shots amongst these is generated.

3.1.4 Combination Shots

Like straight-in and bank shots, the generation of combination shots is straightforward. Working backwards from the object ball to be pocketed, a ghost ball position is found that will send that ball into the desired pocket. This is where the secondary object ball must contact that object ball. A ghost ball position on the secondary object ball can be found by setting the first ghost ball position as that ball's target. The cue ball is then aimed at this secondary ghost ball position.

To generate combination shots, for every pair of object balls, for every pocket, a shot is considered. For the shot to be physically possible, both object balls must be between the cue ball and the target pocket, and the first object ball to be contacted must be closer to the cue ball than the second object ball.

Like bank shots, it is unreasonable to generate a minimum velocity table for combination shots. The initial velocity for each generated shot is set at $2m/s$, sufficient for the vast majority of combination shots.

3.1.5 Shot Difficulty

The difficulty of a straight-in shot is a function of several parameters. A subset of these depend entirely on the position of the object ball and cue ball, independent of the rest of the table state. These are the cut angle α, the object ball-pocket distance $d1$, the cue-object ball distance $d2$, and object ball-pocket angle β (Figure 3.6). α and $d2$ are calculated relative to a ghost ball position. In general, the larger α, $d1$, or $d2$ is, the more difficult the shot. The relationship between β and shot difficulty depends on whether the shot is into a corner or a side pocket.

The shot difficulty function is different between shots into corner and side pockets because of two properties of the geometry of the billiards table. First, the effective pocket size varies differently with the β angle for the corner and side pockets. Effective pocket

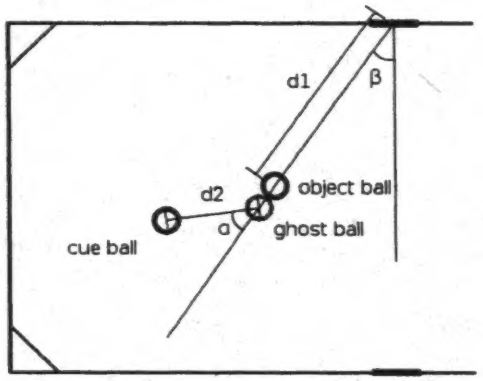

Figure 3.6: Shot difficulty parameters

size refers to the width of the pocket along the trajectory of the object ball. The smaller this is, the less margin for error in the shot. For a side pocket, the pocket is largest, and hence the shot easiest, for a straight-in shot with $\beta = 0$. As β increases, pocket size decreases rapidly. For corner pockets, the pocket is angled[2], so effective pocket size is less sensitive to changes in β. Figure 3.7 illustrates this. $w1$ shows the width of the corner and side pockets where $\beta = 0$, and $w2$ shows the width of the corner and side pockets with a larger β angle. Because of the pocket shapes, the side pocket width is much more sensitive to changes in β. This is compounded by β having a maximum value of 90° at a side pocket, and only 45° at a corner pocket.

The second factor is that the interaction between the object ball and rails differs between the two types of pockets. If an object ball misses the pocket slightly and hits a side rail when aimed at a side pocket, it will never go in. On the other hand, if an object ball hits a rail on the way to a corner pocket, it will still frequently go in, because of the angled pocket orientation. Figure 3.8 illustrates this. This leads to shots along a rail into a corner pocket being much easier than into a side pocket.

Previous work on billiards AI has concentrated on approximating the shot difficulty function to find the easiest shot. Chua et al. used fuzzy logic to do this [7]. In this work, fuzzy sets were defined for $d1$, $d2$, and α corresponding to easy, medium, and hard difficulty values for each parameter (β was ignored). Rules combining these fuzzy variables were

[2]The angled corner pockets used by *poolfiz* approximate the effect of the jaws of the corner pockets on a physical billiards table.

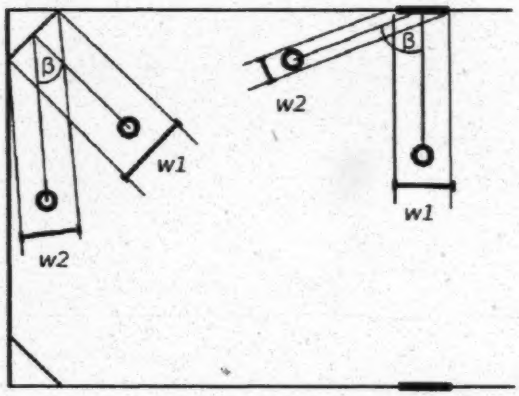

Figure 3.7: Effect of β on effective pocket size

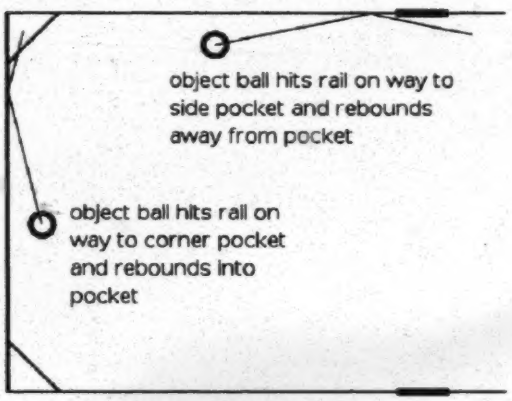
object ball hits rail on way to side pocket and rebounds away from pocket
object ball hits rail on way to corner pocket and rebounds into pocket

Figure 3.8: Rail interactions with corner and side pockets

then used to assign an overall difficulty to the shot. Their evaluation of this approach consisted of the chosen shots being deemed 'acceptable' by a human.

A similar approach was used by Alian and Shouraki [5]. Whereas the rules combining fuzzy variables used by Chua *et al.* were hand-crafted, this work used a reinforcement learning algorithm to tune them on a simulated table. Later work by Alian *et al.* [6] experimented with replacing the reinforcement learning algorithm with a genetic algorithm; this led to worse performance. Evaluation of these approaches was again made by comparing the shots chosen by the technique to ones chosen by a human for the same table state. Every shot chosen under the reinforcement learning-trained algorithm was deemed 'acceptable', and most matched the shot 'recommended' by the human. The genetic algorithm-trained variant chose fewer of these shots, and more that were 'not accepted'.

Lin *et al.* used grey logic to create a mathematical function approximating the shot difficulty function [8]. The parameters $d1$, α, and β were used as inputs. Experiments on a physical robot were used to evaluate this approach. While pictures of this experiment in execution were presented, numerical results of its outcome were not.

PickPocket uses a different approach from these previous methods, taking advantage of the *poolfiz* simulator to capture the shot difficulty function in a table. The table is filled with accurate approximations of absolute probability values. Previous techniques generated arbitrary, relative values.

PickPocket precomputes a table to capture the difficulty function, as the calculation is too costly to perform for each shot generated at runtime. The shot difficulty parameters are discretized and sampling is used to fill each table entry. For each set of parameter values $\{\alpha, d1, d2, \beta\}$, a table state and shot are generated. The shot is simulated s times in *poolfiz*, and the percentage of these that the ball is successfully pocketed is recorded in the table.

At runtime, a table lookup is made for each generated straight-in shot by finding the nearest table entry to the actual difficulty parameters for the shot. Each parameter is rounded to the granularity of the discretization used for the table. The value corresponding to these parameters is retrieved from the table, providing a quick estimate of the shot's probability of success.

In a game situation, the actual probability of success of a shot depends on dynamic factors that cannot be captured in this lookup table. Other object balls on the table can

interfere with the shot when they are near the intended trajectories of the object and cue ball. Slight deviations from these trajectories that would be insignificant on a clear table can now result in collisions with these obstacle balls. The exact value of the five shot parameters also has a small effect on the chance of success.

The granularity of the discretization used for the table has an impact on its accuracy, as well as its memory footprint and computation time. Thus it must be chosen with these factors in mind. PickPocket builds two $30 \times 30 \times 30 \times 30$ tables, one for the corner pockets and one for the side pockets, because of their differing difficulty functions. PickPocket uses $s = 200$, chosen for sub-24 hour precomputation time at the table granularity used.

Bank, kick, and combination shots have too many parameters to construct a success probability table of manageable size. For each collision there is an additional distance and angle parameter. To assign a probability value to these shots, each one is mapped to a corresponding straight-in shot. A discount factor is applied to the straight-in probability to account for the additional complexity of the shot class. This is an approximation that captures the increased difficulty of these shot classes, but does not have the high degree of accuracy of straight-in probability table lookups.

3.1.6 Velocity Table

To quickly find the minimum velocity required for a straight-in shot to sink the target object ball, a table of minimum velocities is precomputed. The minimum velocity for a straight-in shot depends on its $d1$, $d2$, and α parameters. The larger the distances $d1$ and $d2$ are, the harder the cue ball must be initially hit to cover those distances. The larger α is, the less energy is transferred from the cue ball to the object ball in the collision, so the harder it must be hit initially to impart sufficient velocity on the object ball to reach the pocket. Note that β and whether the shot is into a corner or side pocket have no impact on the minimum velocity needed to reach the pocket.

Like the shot difficulty table, to build the velocity table $d1$, $d2$, and α are discretized and a table state and shot are generated for each combination of these parameters. Initially the velocity of the shot is set to $0.1m/s$. It is increased in $0.1m/s$ increments and sampled (without applying the error model) until the target object ball is successfully pocketed. This is the minimum velocity for that set of parameters, and fills that entry in the table.

At runtime, a table lookup given a shot's parameters quickly retrieves the approximate minimum velocity for that shot. Any velocity below the minimum is insufficient to pocket the target object ball, and not worth considering. Larger velocities will still sink the target object ball, but will lead to a different final cue ball position.

PickPocket uses an $80 \times 80 \times 135$ minimum velocity table.

In practice, a small boost of $0.2m/s$ is added to the values retrieved from the velocity table to ensure the target object ball is hit hard enough to sink it. This boost is enough to cover the uncertainty that arises from perturbations to the requested shot velocity by the error model, as well as the error introduced by rounding the shot parameters to the granularity of the lookup table.

3.1.7 Safety Shots

With a safety shot, the goal is not to sink a ball, but rather to leave the opponent with no viable shots. Ideally the opponent will then give up ball-in-hand, leaving the player in a strong situation. Unlike the previously discussed shot classes, there is no way to generate a safety directly from parameters. What makes a good safety is wholly dependent on the table state. The goal is abstract (leave the table in a 'safe state') rather than concrete (sink ball *x* in pocket *y*).

One way to account for the entire table state is the use of sampling. For safety shots, a wide range of ϕ and V values are sampled, leaving θ, a, and b alone to make the sampling space manageable. For each set of ϕ and V, a shot with these parameters is sampled i times, evaluating the resulting state from the opponent's perspective. The overall value of this shot is then the average of these evaluations. i is set equal to the $num\_samples$ parameter from Section 3.3.2, typically to a value of 15.

Since sampling is a costly operation, if safeties were generated per-node then the cost of searching would quickly become excessive. To get around this, safety shots are only considered at the root. Sampling as a one-time cost has a relatively minor impact on performance. At the root, if the best shot from the search has a score below a threshold t_0, safety shots are generated. If the value for the opponent of the best safety is below another threshold t_1, this shot is selected instead of the search result shot. The thresholds t_0 and t_1 can be adjusted to alter the program's safety strategy. For the 10^{th} Computer

Olympiad, these values were set at $t_0 = 0.65$ and $t_1 = 0.5$ (evaluation values range from 0 to 1.48). Later experimentation suggested that this chose safeties too frequently; a strategy that played fewer safeties was more successful (see Section 4.3.3). For the 11^{th} Computer Olympiad, PickPocket used $t_0 = 0.5$ and $t_1 = 0.18$, which fared better in tests.

3.1.8 Bank/Kick/Combo Activation

Occasionally there are straight-in shots available in a table state, yet a bank, kick, or combination shot would actually be a better option for the player. If these shot classes were only considered when there are no straight-in shots available, such opportunities would be missed. Generating all of these shots every time 'just in case' is clearly excessive; it would blow up the branching factor while the vast majority of the time a straight-in shot will be the best shot.

To consider these shots only when they may be of benefit, a threshold similar to that used for safety shots is employed. If the best shot found by the search has a value below b_0, bank, kick, and combination shots are generated and searched. This extra search is only performed to 1-ply, to minimize the effect on average time for shot selection. If bank shots are found that have a greater score than the best shot found by the initial search, the initial search of straight-in shots is repeated to a 1-ply depth; values returned from different depth searches are not directly comparable, as values propagated up more ply are likely to be lower. If the bank/kick/combo shot score is still higher than the score of the best shot in the 1-ply re-search, this shot is executed instead of the shot initially found by the search. This approach has a minimal computational overhead, while still finding bank/kick/combo shots on the rare occasions that they are better options than a straight-in shot. PickPocket uses $b_0 = 0.8$, a hand-tuned value selected so that this feature would kick in when it has a chance of finding a better shot, yet not activate excessively slowing down the program's execution.

3.1.9 Root Sampling

The root node of a billiards search tree has two special features. First, it is the only node in that tree that is guaranteed to be seen in practice, since it is the current table state at the time of search. The player will never be faced with any of the other exact states that comprise

the search tree, only similar states to them. Second, operations performed only at the root node have a one-time cost, as opposed to a per-node cost. Thus it is possible to invest some time improving the average quality of shots available at the root. These improved shots are directly applicable to the current state. There is less benefit to improving the shots available deeper in the tree, as the opportunity to actually execute those shots would never come up.

The number of variations applied to the V, a, and b parameters of each generated shot determines the branching factor of the search algorithm. It also determines the number of options for position play that are available, as each set of parameters will leave the cue ball in a different final position. The more shot variants generated, the better the options for position play, but the longer searching will take. Typically PickPocket generates 3-8 variants of any straight-in shot, to keep the branching factor manageable.

At the root, many more variants can be generated, 50-100 per shot, and each variant can be sampled repeatedly, similar to safety shot sampling. The evaluation function is used to score each shot instance. The averages of these evaluations can be sorted, and the shot variants with the highest scores under sampling can be passed on to the search algorithm. This finds the shot variants that are most likely to be successful, and lead to the best position. This gives the search algorithm a better set of shots to search than the method of 'blindly' varying the shot parameters. Because extensive per-node sampling has a high overhead, this is only feasible at the root as a one-time cost per search.

Viewed another way, this is analogous to iterative deepening on the first ply only. A broad 1-ply sampling search is used to find a smaller set of the best candidates to use in the deeper search.

3.2 Evaluation Function

An evaluation function generates, for a game state, a value corresponding to the worth of that state for the player to act. In search, the evaluation function is applied to the game states at the leaf nodes, and the generated values are propagated up the tree.

In PickPocket's billiards evaluation function, the value of a state is related to the number and quality of the shots available to the player. This is similar to the mobility term used in games like chess, extended to account for the uncertainty of the stochastic domain.

Intuitively, the more high success probability shots available to the player, the more ways he can clear the table without giving up the turn. Even if there is no easy path to clearing the table, the more good options the player has, the greater the chances are that one of them will leave the table in a more favourable state. Similarly, the higher the probability of success of the available shots, the more likely the player is to successfully execute one and continue shooting. Unintentionally giving up the shot is one of the worst outcomes in all billiards games.

To implement this term, the move generator is used to generate shots for the state being evaluated. These shots are sorted by their success probability estimates, highest first. Duplicate shots for the same ball on the same pocket are eliminated, as these all have the same estimate. The first n shots are considered, and the function $d_1p_1 + d_2p_2 + d_3p_3 + ... + d_np_n$ is applied. d_n is the discount factor for the nth shot and p_n is the estimated probability for the nth shot. Values are chosen for each d_n such that they decrease as n increases.

The discount factor is applied to account for diminishing returns of adding additional shots. Consider two situations for a state: three shots with 90%, 10%, 10% success chances, and three shots with 70%, 70%, and 70% chances. These are of roughly equal value to the player, as the former has an easy best shot, whereas the latter has several decent shots with more options for position play. With equal weighting, however, the second would evaluate to nearly twice the value of the first state. Applying a discount factor for shots beyond the first maintains a sensible ordering of evaluations. PickPocket uses $n = 3$, $d_1 = 1.0$, $d_2 = 0.33$, and $d_3 = 0.15$. These weights have been set manually, and could benefit from tuning via a machine learning algorithm.

Another possibility is to evaluate as the quality of the best shot. That is, with $n = 1$ and $d_1 = 1.0$. Intuitively this loses information compared to the above set of parameters. The more good shots available from a state, the more robust it is; the more likely there are to be good shots available even when the state actually arrived at differs from the state evaluated due to the error introduced when making a shot. In practice, the experiments in Section 4.3.2 suggest that PickPocket's evaluation is not very sensitive to the exact values of its parameters.

3.3 Search Algorithms

A search algorithm defines how moves at a node are expanded and how their resulting values are propagated up the resulting search tree. For traditional games like chess, $\alpha\beta$ is the standard algorithm. For stochastic games, the search algorithm must also account for inherent randomness in the availability or outcome of actions. In billiards, players cannot execute their intended shots perfectly. The outcome of a given shot varies, effectively randomly, based on the accuracy of the shooter. For any stochastic game, the search algorithm should choose the action that has the highest expectation over the range of possible outcomes.

When searching billiards, a physics simulation is used to expand the shots available at a node to the next ply. The per-node overhead of simulation reduces the maximum tree size that can be searched in a fixed time period. Whereas top chess programs can search millions of nodes per second, PickPocket searches hundreds of nodes per second.

3.3.1 Probabilistic Search

Expectimax, and its *-Minimax optimizations, are natural candidates for searching stochastic domains [15]. In Expectimax, chance nodes represent points in the search where the outcome is non-deterministic. The value of a chance node is the sum of all possible outcomes, each weighted by its probability of occuring. This approach does not apply directly to billiards, as there is a continuous range of possible outcomes for any given shot. The chance node would be a sum over an infinite number of outcomes, each with a miniscule probability of occuring. To practically apply Expectimax, similar shot results have to be abstracted into a finite set of states capturing the range of plausible outcomes. In general, abstracting billiards states in this way is a challenging unsolved problem.

A simple abstraction that can be made, however, is the classification of every shot as either a success or failure. Either the target object ball is legally pocketed and the current player continues shooting, or not. From the move generator, p_s, an estimate of the probability of success, is provided for every generated shot s. Expectimax-like trees can be constructed for billiards, where every shot corresponds to a chance node. Successful shots are expanded by simulation without applying the error model. For a shot to succeed, the

deviation from the intended shot must be sufficiently small for the target ball to be pocketed, so the outcome table state under noisy execution should be similar to the outcome under perfect execution. For unsuccessful shots, there is no single typical resulting state. The deviation was large enough that the shot failed, so the table could be in any state after the shot. To make search practical, the value of a failed shot is set to zero. This avoids the need to generate a set of failure states to continue searching from. It also captures the negative value to the player of missing their shot.

Unlike games such as chess where players strictly alternate taking moves, billiards has an open ended turn structure. A player may continue shooting as long as they legally pocket object balls. They only give up the shot when they miss, or call a safety. Because the table state after a failed shot is unknown, it is not possible to consider the opponent's moves in search. Thus, this search method only considers the shots available to the shooting player. The goal is to find a sequence of shots which is likely to clear the table, or leave the player in a good position from which to clear the table.

Probabilistic search, an Expectimax-based algorithm suitable for billiards, is shown in Figure 3.9. It has a `depth` parameter, limiting how far ahead the player searches. `Simulate()` calls the physics library to expand the shot, without perturbing the requested shot parameters according to the error model. `ShotSuccess()` checks whether the preceding shot was successful in pocketing a ball.

There are three main drawbacks to this probabilistic search. First, the probability estimate provided by the move generator will not always be accurate, as discussed earlier. Second, not all successes and failures are equal. The range of possible outcomes within these two results is not captured. Some successes may leave the cue ball well positioned for a follow-up shot, while others may leave the player with no easy shots. Some failures may leave the opponent in a good position to run the table, whereas some may leave the opponent with no shots and likely to give up ball-in-hand. Third, as the search depth increases, the relevance of the evaluation made at the leaf nodes decreases. Expansion is always done on the intended shot with no error. In practice, error is introduced with every shot that is taken. Over several ply, this error can compound to make the table state substantially different from one with no error. The search depth used for the experiments in Chapter 4 was restricted more by this effect than by any time constraints relating to tree

```
float Prob_Search(TableState state, int depth){

  // if a leaf node, evaluate and return
  if(depth == 0) return Evaluate(state);

  // else, generate shots
  shots[] = Move_Generator(state);

  bestScore = -1;
  TableState nextState;

  // search each generated shot
  foreach(shots[i]){
    nextState = Simulate(shots[i], state);
    if(!ShotSuccess()) continue;
    score = shots[i].probabilityEstimate
            * Prob_Search(nextState, depth - 1);
    if(score > bestScore) bestScore = score;
  }

  return bestScore;
}
```

Figure 3.9: Probabilistic search algorithm

size. The player skill determines the magnitude of this effect.

3.3.2 Monte-Carlo Search

Sampling is a second approach to searching stochastic domains. A Monte-Carlo sampling is a randomly determined set of instances over a range of possibilities. Their values are then averaged to provide an approximation of the value of the entire range. Monte-Carlo techniques have been applied to card games including bridge and poker, as well as board games such as go. The number of deals in card games and moves from a go position are too large to search exhaustively, so instances are sampled. This makes the vastness of these domains tractable. This suggests sampling is a good candidate for billiards.

In PickPocket, sampling is done over the range of possible shot outcomes. At each node, for each generated shot, a set of *num_samples* instances of that shot are randomly perturbed by the error model, and then simulated. Each of the *num_samples* resulting table states becomes a child node. The score of the original shot is then the average of the scores of its child nodes. This sampling captures the breadth of possible shot outcomes. There will be some instances of successes with good cue ball position, some of successes with poor position, some of misses leaving the opponent with good position, and some of misses leaving the opponent in a poor position. Each instance will have a different score, based on its strength for the player. Thus when these are averaged, the distribution of outcomes will determine the overall score for the shot. The larger *num_samples* is, the better the actual underlying distribution of shot outcomes is approximated. However, tree size grows exponentially with *num_samples*. This results in searches beyond 2-ply being intractable for reasonable values of *num_samples*.

Figure 3.10 shows pseudo-code for the Monte-Carlo approach. `PerturbShot()` randomly perturbs the shot parameters according to the error model.

Generally, Monte-Carlo search is strong where probabilistic search is weak, and vice versa. Monte-Carlo search better captures the range of possible outcomes of shots, but is limited in search depth. Probabilistic search generates smaller trees, and therefore can search deeper, at the expense of being susceptible to error.

Note that in the case where there is no error, probabilistic search and Monte-Carlo search are logically identical. Searching to a given search depth, they will both generate

```
float MC_Search(TableState state, int depth){

  // if a leaf node, evaluate and retuern
  if(depth == 0) return Evaluate(state);

  // else, generate shots
  shots[] = Move_Generator(state);

  bestScore = -1;
  TableState nextState;
  Shot thisShot;

  // search each generated shot
  foreach(shots[i]){
    sum = 0;
    for(j = 1 to num_samples){
      thisShot = PerturbShot(shots[i]);
      nextState = Simulate(thisShot, state);
      if(!ShotSuccess()) continue;
      sum += MC_Search(nextState, depth - 1);
    }
    score = sum / num_samples;
    if(score > bestScore) bestScore = score;
  }

  return bestScore;
}
```

Figure 3.10: Monte-Carlo search algorithm

the same result. It is entirely in how they handle the uncertainty introduced by the error model that the two algorithms diverge.

3.3.3 Search Enhancements

Both probabilistic and Monte-Carlo search algorithms can be optimized with $\alpha\beta$-like cutoffs. By applying move ordering, sorting the shots generated by their probability estimate, likely better shots will be searched first. Cutoffs can be found for subsequent shots whose score provably cannot exceed that of a shot already searched. This reduces the total number of nodes expanded, lowering search times. The search result is not impacted, as only branches that provably could not return the best score are pruned.

For probabilistic search, the pruning test is whether the probability estimate of the current shot multiplied by the maximum evaluation value provably makes that shot inferior to the best shot already found. There are two possible cases for this best shot: either it is an earlier child of the current node, or it is an earlier child of a parent node. If it is an earlier child of the current node, the cutoff check is simply whether the maximum evaluation times the probability estimate of the current shot is less than the score of that earlier child. If this is true, searching the current shot cannot possibly yield a better score, so it can be skipped. If the best shot is an earlier child of a parent node, the cutoff check is whether the current shot times the maximum evaluation *could* exceed the score of the best shot after being propagated up the tree - the probabilities of success of the parent nodes need to be taken into account, as they will be multiplied in as the result for this node is propagated upwards. If the current shot provably could not exceed that best score when propagated up, a cutoff can be made. Since the shots are sorted in order of descending probability estimates, whenever a cutoff is found the cutoff condition will also be true for all subsequent shots. Therefore, searching for that node is finished. Figure 3.11 shows pseudocode for probabilistic search with pruning.

At the root the *cutoff_threshold* parameter is initialized to -1, as there can be no cutoffs until at least one leaf has been evaluated (since the maximum possible score for a child node will never be less than a negative value). The *cutoff_threshold* parameter is divided by shot difficulty estimates as the tree is expanded to reflect the fact that the score at any node will be multiplied by the probability of success of the shot that led to it. For example, if the

```
float Prob_Search(TableState state, int depth,
                  double cutoff_threshold){

  if(depth == 0) return Evaluate(state);
  shots[] = Move_Generator(state);

  double bestScore = -1;
  TableState nextState;

  foreach(shots[i]){
    // test for cutoff
    if(shots[i].probabilityEstimate * MAX_EVAL <
                  cutoff_threshold) break;

    nextState = Simulate(shots[i], state);
    if(!ShotSuccess()) continue;
    score = shots[i].probabilityEstimate
          * Prob_Search(nextState, depth - 1,
                  cutoff_threshold /
                  shots[i].probabilityEstimate);
    if(score > bestScore) bestScore = score;

    // update cutoff threshold
    if(score > cutoff_threshold) cutoff_threshold = score;
  }

  return bestScore;
}
```

Figure 3.11: Probabilistic search w/ pruning

best score so far at a node is 0.6, and the probability of success of the next shot to search is 0.8, then that next child node must yield a score of at least $0.6/0.8 = 0.75$ in order to score higher than 0.6 when multiplied by its probability of success. If, at that next child node, the best achievable score is provably under 0.75, that child node can be pruned.

For Monte-Carlo search, the pruning test is whether the current shot being sampled could exceed the best score so far if every remaining sample yielded the maximum evaluation. If it could not, the remaining sampling for the shot can be skipped. The check for this condition can be made after every sample is taken. Subsequent shots from that same node still need to be sampled, as they could still potentially yield a higher value. This is

different from probabilistic search, where once a cutoff is found, search for the entire node is complete. However, because shots are searched in order of descending probability estimates, it is likely that cutoffs will be found for the later shots at a node. Figure 3.12 shows pseudocode for Monte-Carlo search with pruning.

Again, the *cutoff_threshold* parameter is initialized to -1 so no cutoffs will be found until at least one leaf has been evaluated. As the tree is expanded, *cutoff_threshold* is propagated from parent to child nodes. Since the actual best score for a parent node is unknown at the point where *cutoff_threshold* is propagated (as it depends on all samples, and not all samples have completed yet), the minimum required value for the child node for the current shot to exceed the best score is propagated. This minimum required value is the value required of the current sample, assuming all future samples evaluate to the maximum possible score. If the child node provably cannot exceed this value, it can be pruned.

3.4 Game Situations

To play billiards games, an AI needs routines to handle the break shot and ball-in-hand situations that occur regularly. This section describes the approach PickPocket uses for these situations.

3.4.1 Break Shot

Every billiards game begins with a break shot. This establishes the position of the object balls on the table, as well as which player gets to continue shooting. In most billiards games, including 8-ball, if a player pockets an object ball on the break shot, they may continue shooting. If they do not, their opponent gets the next shot.

Poolfiz randomizes the size of the small spaces between the object balls in the initial rack, leading to variation in the outcome of the break shot. Thus, break results are unpredictable. It is not feasible to respond dynamically to the exact details of the initial rack. The player can, however, select a shot that maximizes their chances of sinking a ball over the range of possible racks.

PickPocket uses sampling to precompute a break shot. A range of:

```
float MC_Search(TableState state, int depth,
                double cutoff_threshold){

  if(depth == 0) return Evaluate(state);
  shots[] = Move_Generator(state);

  double bestScore = -1;
  TableState nextState;
  Shot thisShot;

  foreach(shots[i]){
    sum = 0;
    for(j = 1 to num_samples){
      // test for cutoff
      if(((sum + (num_samples - j + 1) * MAX_EVAL)
            / num_samples) < cutoff_threshold) break;

      thisShot = PerturbShot(shots[i]);
      nextState = Simulate(thisShot, state);
      if(!ShotSuccess()) continue;
      sum += MC_Search(nextState, depth - 1,
                       cutoff_threshold * num_samples
                       - sum - ((num_samples - j)*MAX_EVAL));
    }
    score = sum / num_samples;
    if(score > bestScore)bestScore = score;

    // update cutoff threshold
    if(score > cutoff_threshold) cutoff_threshold = score;
  }

  return bestScore;
}
```

Figure 3.12: Monte-Carlo search w/ pruning

- Initial cue ball positions,[3]
- Velocities, and
- ϕ aiming angles

are sampled, with 200 samples taken for each set of parameters. The percentage of these that manage to sink an object ball is recorded. After sampling all positions, the set of parameters the led to the highest chances of sinking a ball are selected. At runtime, when it is PickPocket's turn to break, this break shot is executed.

3.4.2 Ball-in-Hand

A billiards player gets ball-in-hand when their opponent commits a foul, or fails to execute a legal shot. In 8-ball, a foul occurs if the cue ball is pocketed, or a player fails to hit a legal ball and rail on their shot. When a player has ball-in-hand, they are free to place the cue ball anywhere on the table. Ball-in-hand is a very strong situation, as the player can take advantage of it to set up an easy shot. Strong players often use ball-in-hand to sink 'trouble' balls that would be difficult to pocket from many positions on the table.

PickPocket must choose a position for the cue ball, as well as select a shot from that position, when it is awarded ball-in-hand. Although the cue ball could be placed anywhere on the table, it is impossible to do a full search from every position. Like with shot generation, a set of the most relevant positions for the cue ball must be generated. From each of these, search can proceed as normal by creating a table state with the cue ball in the selected position. The cue ball is ultimately placed at the position that led to the best search result, and the shot selected by that search is executed.

To generate a set of candidate positions, the table is discretized into a grid of cells. Each cell is assigned a value by generating shots as though the cue ball were at the center of that cell. Probability estimates for these shots are retrieved from the probability table. For each cell, the probability estimate of the best shot is set as the value of that cell.

This creates a map of the table, with the value of each cell corresponding to the ease of the best available shot from that cell. From this map, a set of candidate positions for the

[3]The cue ball may be placed anywhere behind the headstring on a break attempt.

cue ball need to be retrieved. These positions should be high valued cells, from a range of different regions on the table to capture the breadth of available options. This is preferred over considering multiple neighbouring cells, as the options available from neighbouring cells are likely to be very similar, and could be captured by evaluating just one of them. To find the highest valued non-adjacent cells, local maxima are examined.

A randomized sampling search is used to approximate the k-best local maxima. A set of c cells on the table are randomly selected, and hill-climbing is performed from each of these to find c local maxima. Duplicates are eliminated. The remaining values are then sorted, and the best k cell locations are returned. These are the candidate positions which are searched to find the final ball-in-hand shot.

Figure 3.13 gives psuedocode for the entire ball-in-hand shot selection process.

Under certain conditions, a player must take their ball-in-hand shot from behind the headstring. In 8-ball this occurs after the opposing player fouls on a break shot. In this case, only grid squares behind the headstring are populated with values. Additionally, in this case the player must shoot forward, aiming at a ball on the far side of the headstring. A flag is set so that only shots aimed at balls on the far side of the headstring are considered during search.

```
Shot Compute_Ball_In_Hand_Shot(TableState state){

  // populate grid
  foreach(grid cell [x,y]){
    state.Cue_Position = cells(x,y).centre;
    ShotSet shots[] = Generate_Shots(state);
    ShotSet.sort();
    grid[x,y] = shots[0].probability_estimate;
  }

  // find best shot
  candidate_positions[] = Get_KBest_Local_Maxima(grid, k);
  Shot best_shot;
  best_shot.score = -1;
  foreach(candidate_positions[i]){
    shot = search(candidate_positions[i]);
    if(shot.score > best_shot.score){
      best_shot = shot;
    }
  }

  return best_shot;
}
```

Figure 3.13: Ball-in-hand algorithm

Chapter 4

Experimental Results

Pickpocket has a wide range of adjustable parameters and features. This chapter presents experimental results demonstrating the impact of these features on the program's performance. Experiments were performed with probability table size, evaluation function parameters, safety shots, root sampling, bank/kick/combination shots, Probabilistic search parameters, Monte-Carlo search parameters, pruning enhancements, and a comparison of the search algorithms. A description of the experimental setup and the results of these tests follow. This chapter also contains the details of the 10^{th} and 11^{th} Computer Olympiad simulated 8-ball tournaments won by PickPocket.

4.1 Error Model

Although the results of shots on a physical table are stochastic, simulator results are deterministic. To capture the range of shot results on a physical table, a random element is introduced into the simulation. In *poolfiz*, error is modeled by perturbing each of the five input shot parameters by zero-mean Gaussian noise. A set of standard deviations $\{\sigma_\phi, \sigma_\theta, \sigma_V, \sigma_a, \sigma_b\}$ corresponding to the noisiness of the five parameters is specified. These σ values can be chosen with the properties of the player being simulated in mind. For a robot, σ values can be approximated experimentally.

The use of Gaussian noise is a simplification. It may well be that on a robot, due to biases of the system, shot errors are not normally distributed. The noise between the desired shot and the actual executed shot on a robot is a result any imprecisions in the robot's vision and control systems. The shot result is also affected by the properties of

the physical table being played on, as well as environmental factors like temperature and humidity. If the actual distribution of shot errors on a robot were to be measured, then this distribution could be used to perturb shots in *poolfiz* instead; there is no reason why the noise model need be Gaussian. *Poolfiz* with its noise model acts as a black box: the user submits their requested shot, it is perturbed by the noise model used, and a resulting table state is returned. The calling program need not know anything about the noise model used internally, so it can be as simple or complex as necessary. Gaussian noise is convenient for its simplicity and as an approximation how actual noise on a robot may appear.

4.2 Sample Size and Statistical Significance

Because of the stochastic nature of the domain, all experimental results are subject to uncertainty. This arises from the two sources of randomness: the random spacing between the balls as they are racked, and the error added to requested shot parameters by the error model. The former leads to variations in the positions of object balls on the table after the break, and the latter leads to variations in the outcome of each requested shot. In the long run - over a very large or infinite number of games - the effects of this randomness even out as both sides are helped and hurt by it equally. However, over the course of tens or hundreds of games, the random element is significant and must be taken into account.

Any pair of billiards programs will have a constant, 'true' underlying win rate between them. This is the proportion of games that each will win over an infinite sample size. The exact value of this win rate depends on the programs' parameters, as well as the σ values that make up the error model. The win rate can also be viewed as the equity that each program has in a match before it begins. If a program is played against itself, each instance of that program will win 50% of the total games in the long run.

The purpose of running matches as experiments is to determine this underlying win rate between program variants, which is a measure of their relative strengths. Since only a finite number of games are played, each match result is an approximation of the underlying win rate. The impact of randomness keeps it from being exact. Ideally we would like to know which side is superior and by how much when comparing programs; we would like to know the exact value for the underlying win rate. However, the sample size required to

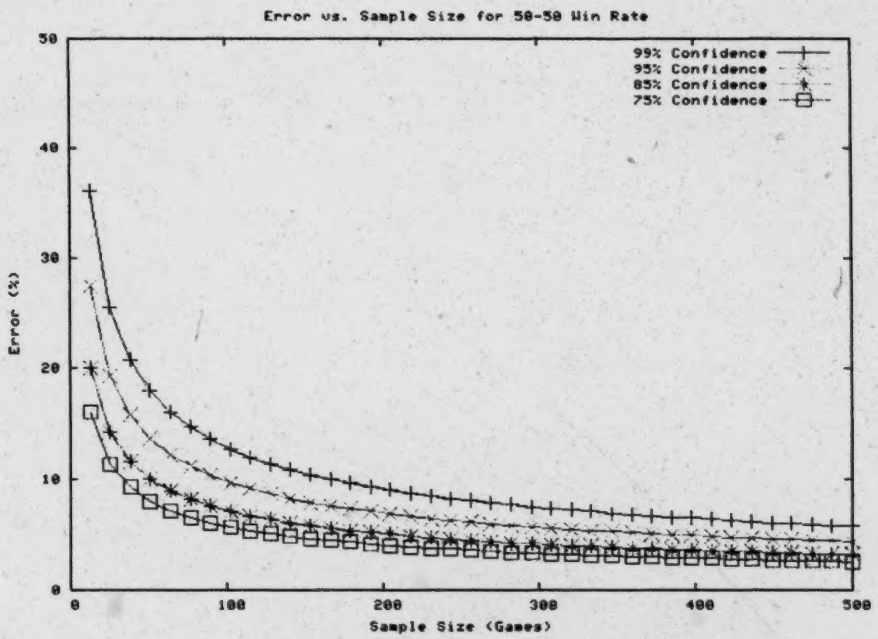

Figure 4.1: Error vs. Sample Size for 50-50 Win Rate

approximate this to a high degree of accuracy would be very large. The experiments in this section will give an indication whether one side is superior to another, or whether they are roughly equal, but the exact underlying win rates will remain unknown.

Statistics provides a formula for relating sample size (number of games in a match), confidence level, and uncertainty:

$$n = \frac{z^2 pq}{E^2} \tag{4.1}$$

where n is the sample size, z is the z-value for the confidence interval (typically retrieved from a table of z-values), p is the underlying win rate for one program, and q is $(1-p)$, or the underlying win rate of the other program. Since these underlying win rates are unknown, in practice the sampled win rates can be filled in - for large enough sample sizes this is sufficiently accurate. E is the magnitude of the error. Therefore, the associated confidence interval is $p \pm E$.

Figure 4.1 shows the relationship between sample size and error for an underlying 50-50 win rate. This clearly shows diminishing returns as the sample size is increased - addi-

Sampled Result	Confidence Level			
	99%	95%	85%	75%
50-50	(37.1-62.9 , 62.9-37.1)	(40.2-59.8 , 59.8-40.2)	(42.8-57.2 , 57.2-42.8)	(44.3-55.7 , 55.7-44.3)
55-45	(42.2-57.8 , 67.8-32.2)	(45.3-54.7 , 64.7-35.3)	(47.8-52.2 , 62.2-37.8)	(49.3-50.7 , 60.7-39.3)
60-40	(47.4-52.6 , 72.6-27.4)	(50.4-49.6 , 69.6-30.4)	(52.9-47.1 , 67.1-33.9)	(54.4-45.6 , 65.6-34.4)
65-35	(52.7-47.3 , 77.3-22.7)	(55.7-44.3 , 74.3-25.7)	(58.1-41.9 , 71.9-28.1)	(59.5-40.5 , 70.5-29.5)

Table 4.1: Confidence Intervals for 100-Game Match Result

tional games lead to progressively smaller reductions in the magnitude of error. 100-game samples were chosen for the experiments in this section to provide a balance between accuracy and execution time. To substantially increase the accuracy of the experiments, a significantly larger sample size would be required.

Table 4.1 shows the confidence intervals for a range of match results and confidence levels, for a 100-game sample size. These show that a 50-50 result or a 55-45 result is inconclusive for determining which side is superior. Underlying win rates favouring both sides fall within their respective confidence intervals. Therefore it is impossible to strongly conclude whether one program will prove superior, or whether they will prove roughly equal, in the long run. A 55-45 result does suggest that the side scoring 55 is stronger, as the 75% confidence interval almost entirely contains scores favouring this side. However, 75% confidence is weak and inconclusive - a larger sample size would be required to provide strong evidence of the long-run result.

A 60-40 result is more conclusive, as the entire confidence intervals up to the 95% confidence level contain scores favouring the 60 side. Therefore, as a general rule of thumb any 100-game experiment where one side wins 60 games or more is strong evidence (over 95% confidence) of that side's superiority. Matches with results closer to 50-50 are not conclusive one way or another to a strong degree of confidence. However, the higher above 50 a score gets, the more likely it becomes that it is the long-run winning program.

4.3 Experiments

PickPocket plays 8-ball, the game selected for the first computational billiards tournament. The rules were described in Section 2.4. To summarize, each player is assigned a set of seven object balls: either solids or stripes. To win, the player must pocket their entire set, followed by the 8-ball. If a player's shot pockets the 8-ball prematurely, they suffer an

automatic loss. Players must call their shots by specifying which object ball they intend to sink in which pocket. A player continues shooting until they fail to legally pocket a called object ball, or until they declare a safety shot.

A series of matches were played to evaluate the impact of PickPocket's many parameters on its performance. Typically each match holds all but one parameter constant, to isolate the effect of that one parameter. Bank, kick, and combination shot generation is disabled, to simplify the experiments and compare purely the effects of search. Therefore, in these matches every shot is either a straight-in attempt, a safety attempt, or a break shot to begin a new game. Players alternate taking the break to eliminate any potential advantage or disadvantage of going first. For each match, the following results are given:

- W (Wins), the number of games the program won.
- SIS (Straight-in-success), the ratio of successful to attempted straight-in shots. A successful straight-in shot sinks the called object ball; the player continues shooting.
- SS (Safety success), the ratio of successful to attempted safety shots. A successful safety is one where the opponent fails to pocket an object ball on their next shot, thus the player gets to shoot again.

Unless otherwise stated, PickPocket is configured as the following for the matches: A $30 \times 30 \times 30 \times 30$ granularity table is used. Monte-Carlo search is used to 2-ply, with $num\_samples = 15$. Safety thresholds are set at $t_0 = 0\%$ and $t_1 = 1.48$, so safeties are played if and only if no other shots can be generated. Enhancements such as Root Sampling and extra Bank/Kick/Combo activation are disabled.

Parameters were chosen such that a decision was made for each shot within approximately 60 seconds. This is a typical speed for time-limited tournament games. A 10-minute per side per game hard time limit was imposed, to ensure that program variants made their decision within a reasonable amount of time. 2-ply Monte-Carlo search was the only variant that approached this limit.

Experiments were run primarily under the E_{Taiwan} error model, with parameters: {0.185, 0.03, 0.085, 0.8, 0.8}. This was the error model used at the 10^{th} Computer Olympiad in Taiwan. It models a strong amateur, who can consistently pocket short, easy shots, but

sometimes misses longer shots. Tests showed than under E_{Taiwan}, 69.07% of straight-in shots in random table states were successful, over a 10,000 shot sample size. These numbers are lower than the sinking percentages seen in experiments with PickPocket, as they are from random table states. In a gameplay situation, PickPocket does not see random table states. Rather, it sees states that are the result of shots chosen partly for their positional value.

Two additional error models were used in the search depth and search algorithm comparison experiments. E_{low}, with parameters {0.0185, 0.003, 0.0085, 0.08, 0.08}, corresponds to a top human player who can consistently make even challenging shots successfully. E_{high}, with parameters {0.74, 0.12, 0.34, 3.2, 3.2}, corresponds to a human novice who can usually make short, easy shots, sometimes make medium difficulty shots, and rarely make challenging shots. Thus, these experiments were repeated under conditions of low, medium, and high error.

4.3.1 Probability Table

Experiments were constructed to demonstrate the effectiveness of the probability table used to estimate shot difficulties. This accuracy is related to the granularity of the discretization used, which determines the number of entries in the table and the average distance between sets of shot parameters and the nearest probability table entry.

To find out how the accuracy of the probability table varies with table size, a tester program was created. This generates random positions with the cue ball and a single object ball on the table. For each position, every physically possible straight-in shot is examined. Each of these shots is sampled under the error model to estimate its probability of success directly. This is then compared to the probability estimate provided by a table lookup for that same shot. The magnitude of the difference between these values is the error for that particular shot. These errors are averaged over all shots to give an indication of the accuracy of the probability table as a whole.

For these experiments, 5,000 positions were examined and 500 samples per shot were taken. Table 4.2 shows the average error reported by this tester program for a variety of table sizes. The Total Shots and Error columns give the shots and average error for each entire table as a whole. The 80%+ columns give the number of shots and error for just

Table Size	Total Shots	Error	80%+	80%+ Err	90%+	90%+ Err
$5 \times 5 \times 5 \times 5$	9142	0.247	529	0.180	77	0.160
$10 \times 10 \times 10 \times 10$	9168	0.177	1164	0.149	703	0.140
$15 \times 15 \times 15 \times 15$	9221	0.152	1440	0.136	852	0.117
$20 \times 20 \times 20 \times 20$	9268	0.147	1427	0.135	935	0.118
$25 \times 25 \times 25 \times 25$	9199	0.146	1476	0.125	889	0.108
$30 \times 30 \times 30 \times 30$	9208	0.142	1437	0.120	900	0.104

Table 4.2: Probability table accuracy

those shots whose probability of success is reported by the probability table as being 0.80 or more. The 90%+ columns give the details for those shots whose probability of success according to the probability table is 0.90 or more.

Not surprisingly, these results clearly show that higher granularity tables generate more accurate probability estimates. Additionally, they demonstrate the diminishing returns inherent when increasing that table size. The jump in overall accuracy from $5 \times 5 \times 5 \times 5$ to $10 \times 10 \times 10 \times 10$ is greater than the jump from $10 \times 10 \times 10 \times 10$ to $30 \times 30 \times 30 \times 30$. Most of the gains come in at the first few steps, even though each jump adds a larger absolute quantity of entries to the table than the jump before it.

Note that the most accurate probability estimates are for the shots that have the highest probability of success according to the table. These happen to be the shots for which accuracy is the most important - the evaluation function used by PickPocket is based on the probability estimate of the best shots, and move ordering during search places the best shots first; weaker shots will often be eliminated by cutoffs.

These entries are the most accurate for two reasons: Firstly, sampling theory indicates that, for a fixed number of samples, the results that are closer to 100%-0% have a lower uncertainty than those close to 50%-50%. Secondly, shots that have a high probability estimate do so because they are robust - changes in the shot input parameters had a minimal effect on the sinking of the shot while the table was being generated. On the other hand, shots with a low probability estimate are not robust. During the table generation, some perturbations of the shot parameters would lead to the shot being a success, and some to it being a failure. Upon table lookup, the actual shot whose probability is being estimated will have different parameters from the shot that was sampled to generate the entry, due to

Match	W	SIS	SS
$5 \times 5 \times 5 \times 5$	38	474/635=74.6%	18/56
$30 \times 30 \times 30 \times 30$	62	546/696=78.4%	21/62

Table 4.3: Probability table match result

the finite granularity of the table. This tends to lead to more variation between the sampled and actual probability in non-robust shots with lower probability estimates, than with the robust, high estimate shots.

Finally, a match was run between two programs that were identical in every parameter except for probability table size. A program using a $30 \times 30 \times 30 \times 30$ table competed against one using a $5 \times 5 \times 5 \times 5$ table. The match results are shown in Table 4.3. The program using the higher granularity probability table sunk 3.8% more of its attempted shots than the program using the lower granularity table. That this led to a 62-38 match win is a convincing, statistically significant evidence that a higher granularity table leads to better performance in match play. It also points out the reality that small improvement in accuracy can lead to a large gain in overall performance.

4.3.2 Evaluation Function

To assess the impact of the evaluation function on performance, games were played between programs using the following evaluation functions:

- Material Difference (MD). The evaluation is the difference between the number of the player's balls on the table and the number of opponent's balls.
- (1,1,1). PickPocket's evaluation with $d_0 = 1$, $d_1 = 1$, and $d_2 = 1$. Thus the evaluation is the sum of the probability table lookup for the best three shots generated.
- (1,0,0). PickPocket's evaluation with $d_0 = 1$, $d_1 = 0$, and $d_2 = 0$. Thus the evaluation is the probability table lookup for the single best shot generated.
- (1,0.33,0.15). PickPocket's evaluation with $d_0 = 1$, $d_1 = 0.33$, and $d_2 = 0.15$.

Note that MD is not a very interesting evaluation for most billiards search trees, as the search is to a fixed depth, and each successful shot typically sinks exactly one object ball.

Match	W	SIS	SS
(1,0,0)	54	574/745=77.0%	23/66
MD	46	503/671=75.0%	25/77
(1,0,0)	53	555/692=80.0%	15/53
(1,1,1)	47	467/615=75.9%	8/46
(1,0,0)	46	543/691=78.6%	22/71
(1,0.33,0.15)	54	522/669=78.0%	18/70

Table 4.4: 2-ply comparison of evaluation functions

Thus the leaf evaluation will differ from the root evaluation by exactly the search depth the vast majority of the time - most leaves will have the same evaluation. The only times the evaluation will differ is when a shot incidentally sinks one or more extra object balls in addition to the called object ball.

The results of these matches, shown in Table 4.4, are all very close to the 50-50 mark, so it is impossible to tell which evaluations are stronger with a high degree of confidence. In fact, this is evidence that the exact composition of the evaluation function is not a dominant parameter for this type of search. MD did lose its match to (1,0,0), and have the weakest shooting percentage, however not by margins large enough to result in high statistical confidence that it is in an inferior evaluation.

To differentiate the evaluations, the matches were re-run with the search depth set equal to one. This places a greater emphasis on the values returned by the evaluation function, as they more directly determine which shot is selected. The evaluation values are only modulated by one ply of search, not two. Here, the variants using weighted probability estimates are still too close to call, however the match between (1,0,0) and MD was decisive. Its 65-35 win indicates that (1,0,0) has the higher winrate of the two programs, with a high degree of confidence. The results of this 1-ply match are shown in Table 4.5.

The likely reason that MD, an evaluation that adds very little information to the search, performed well to a 2-ply depth is that the 2-ply search implicitly generates position play. Any branch at the root that has a high value has two good successive shots available, as well as a high evaluation value. If it did not, the evaluation value would not be propagated up the tree intact - the root would show a lower value for that branch due to the weakness of the available shots. On the other hand, a 1-ply search relies on the evaluation function

Match	W	SIS	SS
(1,0,0)	65	612/766=79.9%	37/101
MD	35	457/624=73.2%	32/107
(1,0,0)	51	546/677=80.6%	34/85
(1,1,1)	49	541/701=77.2%	18/57
(1,0,0)	45	561/705=79.6%	31/77
(1,0.33,0.15)	55	524/678=77.2%	19/68

Table 4.5: 1-ply comparison of evaluation functions

Match	W	SIS	SS
Last Resort	61	555/714=77.7%	49/103=47.6%
Taiwan	39	434/536=81.0%	104/170=61.2%
Last Resort	48	521/654=79.7%	35/85=41.1%
Italy	52	515/646=79.7%	50/86=58.1%

Table 4.6: Safety match result

to provide information to lead to position play. MD does not do this, so leads to positions that are weaker on average than those found by searching with an evaluation related to the number and quality of shots available.

The results also suggest that of the four functions tested, (1, 0.33, 0.15) is the best. Therefore, it was used in all subsequent experiments.

4.3.3 Safety Shots

To evaluate the impact of PickPocket's safety shot activation thresholds on performance, matches were played between the following variants:

- Last Resort. Baseline with safety thresholds set to $t_0 = 0.0$ and $t_1 = 1.48$. Safety shots are played if and only if no other shots are available.
- Taiwan. The safety strategy used at the 10^{th} Computer Olympiad. $t_0 = 0.65$ and $t_1 = 0.50$. This plays safeties aggressively, choosing a moderate scoring safety over a moderate straight-in shot.
- Italy. The safety strategy used at the 11^{th} Computer Olympiad. $t_0 = 0.50$ and $t_1 = 0.18$. While it considers safety shots only slightly less frequently than the

Match	W	SIS	SS
Root sampling	59	567/666=85.1%	17/41
No root sampling	41	426/544=78.3%	10/38

Table 4.7: Root sampling match result

Taiwan strategy, it is much more conservative about executing safety shots. The average evaluation for the opponent of the safety state must by 0.18 or less for a safety to be activated over a weak straight-in shot. For this to be true, the opponent must consistently have no good shot available when the safety is searched.

The results of this match are shown in Table 4.6. The Taiwan strategy played the most safeties, as expected, and lost its match in a statistically significant manner, scoring 39-61. The results of the Italy strategy match are too close to call. It is not clear which is the long-run better strategy between it and the baseline, but the result suggests that they are roughly equal in performance. These results show that playing too many safety shots in 8-ball is a poor strategy, hence PickPocket's safety thresholds were refined for the 11^{th} Computer Olympiad.

Safety shots were 41% to 61% effective overall (SS Column). The Last Resort strategy consistently had the poorest safety success record, as it would only choose safeties when there were no straight-in shots available. In these cases it would execute a safety whether or not a good one was found. The other strategies would only execute safeties when other shots were available when a moderate-to-good safety was found. Therefore, they would be expected to show the better overall success that they did.

4.3.4 Root Sampling

A match was played between a version of the program with the root sampling feature enabled, and one with the root sampling feature disabled. The results of this match are shown in Table 4.7.

Root sampling operates similar to iterative deepening in traditional search, in that it broadly searches 1-ply before doing a deeper search of the best shots. This improved the shooting percentage to 85.1%, whereas Monte-Carlo search without this enhancement has shooting percentages in the 77%-81% range. The improved shooting percentage is

evidence of enhanced position play, as considering more shots at the root of the search tree gives the program more options to find a shot that leads to good position. The 59-41 match result is further strong evidence of the benefit of this enhancement, as it suggests it is the better program to just under 95% confidence.

This shows the importance of search breadth in billiards. Root sampling was set to consider 50 variants of each ball and pocket combination at the root, each variant with slightly different a, b, and V parameters. Regular shot generation finds 3-8 variants per straight-in shot.

4.3.5 Probabilistic Search

Matches were played between variants of the Probabilistic Search algorithm searching to various depths to evaluate the relationship between search depth and performance. The matches were repeated under three different error models: E_{low}, E_{Taiwan}, and E_{high}, corresponding respectively to low, medium and high amounts of noise added to the attempted shots. The results of the matches are shown in Table 4.8.

Under E_{low}, the various search depths all have roughly equal performance. There is no apparent gain from searching deeper, but neither is there a significant penalty. Under E_{Taiwan} and E_{high}, on the other hand, 1-ply search clearly outperforms any deeper search, winning every match under these error models. This suggests that, as error increases, any benefit to deep lookahead is cancelled out by the compounding effect of error over several ply of search. Probabilistic search, with its deterministic node expansion, is expected to be the algorithm most susceptible to this type of error.

Leckie's experiments in [24] show a similar result. He found that, for his Expectimax-based billiards search algorithm, deeper search did improve performance when the amount of error introduced was small. When he repeated the experiments with larger error, deeper search fared worse.

4.3.6 Monte-Carlo Search

Monte-Carlo search is controlled by two parameters: search depth and $num\_samples$, the number of samples to expand for each shot. Table 4.9 shows the effect of sample size in a 1-ply search. Table 4.10 shows the effect of sample size in a 2-ply search. Note that this

Match	W	SIS	SS
	E_{low}		
Depth 1	48	446/502=88.8%	56/101
Depth 2	52	464/519=89.4%	58/99
Depth 1	54	502/557=90.1%	38/86
Depth 3	46	446/523=85.3%	35/69
Depth 1	53	450/507=88.7&	40/78
Depth 4	47	481/532=90.4%	38/86
	E_{Taiwan}		
Depth 1	66	541/698=77.5%	90/164
Depth 2	34	459/661=69.4%	76/137
Depth 1	63	545/687=79.3%	83/160
Depth 3	37	518/689=75.2%	68/142
Depth 1	61	534/708=75.4%	83/141
Depth 4	39	511/697=73.3%	72/139
	E_{high}		
Depth 1	59	475/1009=47.1%	143/219
Depth 2	41	463/1034=44.8%	122/189
Depth 1	52	487/1056=46.1%	156/226
Depth 3	48	478/1098=43.5%	110/186
Depth 1	59	522/1082=48.2%	148/225
Depth 4	41	490/1129=43.4%	96/163

Table 4.8: Effect of search depth in probabilistic Search

Match	W	SIS	SS
Samples = 5	47	492/615=80.0%	25/73
Samples = 15	53	504/637=79.1%	21/59
Samples = 5	46	532/688=77.3%	27/73
Samples = 30	54	547/707=77.3%	29/73

Table 4.9: Effect of sample size in 1-ply Monte-Carlo search

Match	W	SIS	SS
Samples = 5	46	500/643=77.8%	23/75
Samples = 15	54	528/667=79.1%	34/66

Table 4.10: Effect of sample size in 2-ply Monte-Carlo search

experiment was not run for $num\_samples = 30$ because that variant did not run within tournament time constraints. Table 4.11 shows the effect of search depth on performance, with $num\_samples$ fixed at 15. Again, 2-ply was the deepest variant that would run within time constraints. As with Probabilistic search, the error model was varied for the search depth experiments.

All of the match results varying sample size are too close to call. This suggests that the exact parameters used in Monte-Carlo search have a relatively minor impact on performance. Interestingly, all three of the variants using a larger sample size edged out the variants using a smaller sample size. This might suggest that larger sample size confers a slight benefit. This would be expected, as the larger the sample size, the better the underly-

Match	W	SIS	SS
	E_{low}		
Depth 1	36	381/439=86.8%	3/35
Depth 2	64	533/593=89.9%	4/18
	E_{Taiwan}		
Depth 1	55	504/616=81.8%	25/57
Depth 2	45	492/621=79.2%	15/41
	E_{high}		
Depth 1	43	471/985=47.8%	50/97
Depth 2	57	501/1008=49.7%	51/103

Table 4.11: Effect of search depth in Monte-Carlo search

ing distribution of shot outcomes is approximated. Certainly a smaller sample size would not be expected to perform better than a larger sample size. However, because the results are all so close, it could just as easily be a result of noise that the larger sample size won every time; too much should not be read into the result.

The experiment with search depth also had an inconclusive result under the E_{high} and E_{Taiwan} error models. It is not clear whether 2-ply Monte-Carlo search performs better, worse, or approximately equal to 1-ply search under these conditions. The results suggest that they are quite close in performance. Compared to Probabilistic search, Monte-Carlo fares better at deeper depths. This is expected as it better handles the error intruduced at every shot. Under the E_{low} error model, 2-ply is stronger than 1-ply with statistical confidence. This echoes the result seen in the search depth experiments with Probabilistic search, where deeper search exhibited better performance when there was less noise relative to when there was more noise.

2-ply did generally perform fewer safety shots than 1-ply search. Since safeties are only executed when no straight-in shots are available, this means it left itself with no straight-in shots fewer times than 1-ply did. This is an expected benefit of deeper search. However, because the safety shots constitute a small percentage of the overal shots in the match, this factor likely had a minimal effect on the final result.

4.3.7 8.5-ball

The experiments with search depth in 8-ball only showed a benefit to further lookahead beyond 1-ply under the E_{low} error model, and there the benefit was only substantial for the Monte-Carlo search algorithm (there is always a large benefit to searching 1-ply vs. a greedy, non-searching algorithm - see Section 4.3.10). With larger error, in Probabilistic search deeper search actually performed worse than 1-ply search. In Monte-Carlo search, the match results were too close to declare any advantage for 2-ply search over 1-ply, or vice versa. This result is surprising as traditionally search depth is strongly correlated with performance in game-playing programs. Interestingly, deeper search was shown to be beneficial under the E_{Taiwan} error model when using the material difference evaluation, in Section 4.3.2. It implicitly added information that compensated for a weak evaluation function.

There are 3 possible reasons why deeper search might not be so beneficial. Firstly, the evaluation function could be good enough that the 1-ply evaluation consistently gives the best shots the highest value. Such strong evaluations exist for backgammon, a game where relatively little additional strength comes from deeper search [15]. This is unlikely to explain the results seen in 8-ball, as deeper search was clearly beneficial under the E_{low} error model using the Monte-Carlo search algorithm.

Secondly, noise makes the states seen, and therefore the evaluations returned, less accurate as depth increases. Because of the noise model, the root of the search tree is the only state that will actually occur in-game. The other nodes that comprise the trees are states that are likely to be similar to states that will be seen. However, inaccuries can compound over several ply. This is likely why deeper search is actually a worse performer in Probabilistic search. Deeper Monte-Carlo search does not clearly fare worse likely because the algorithm is better suited to accounting for this type of error. As noise increased from E_{low} to E_{high}, both algorithms saw a decrease in the effectiveness of deeper search.

The third possible reason for the lack of benefit to additional search depth has to do with the properties of 8-ball. When playing, the player has the option of shooting at any of their objects balls. At the beginning of the game there are seven balls they can select to shoot at. As the table clears, they have fewer potential object balls to target, but at the same time there are fewer object balls obscuring potential shots. The player will very frequently have shots available. Further, this feature makes position play easier because the player can set up for position on any of their remaining object balls. The potential shots in 8-ball are not very constrained. Because of this, it is likely that there will be good shots available from a wide range of table states. It is hard for a player to run themselves into a corner and be left with no good shots available. This can be seen from the relative infrequency at which safety shots are played in PickPocket's 8-ball matches.

In contrast, some billiards games more strongly constrain the shots available to the player. In 9-ball, the player must shoot at the lowest numbered ball remaining on the table. When shooting, they are aiming at one specific ball, and trying to gain position on one other specific ball (the next-lowest ball remaining on the table), rather than having the option of any ball in a set. From more table states there will not be shots available that both sink the target ball and get position on the next ball. It is easier for a player to run themselves into

Match	W	SIS	SS
Depth 1	44	420/666=63.0%	266/423
Depth 2	56	428/664=64.5%	283/433
Depth 1	65	477/693=68.8%	281/427
Depth 3	35	399/620=64.4%	257/434
Depth 1	56	438/669=65.5%	300/466
Depth 4	44	447/714=62.6%	254/430

Table 4.12: Effect of search depth in Probabilistic 8.5-ball

Match	W	SIS	SS
Depth 1	40	380/509=74.7%	73/139
Depth 2	60	378/517=73.1%	54/123

Table 4.13: Effect of search depth in Monte-Carlo 8.5-ball

a corner in such games. Therefore, deeper search would be expected to confer a stronger benefit in these games.

To test whether this is the case, the game of 8.5-ball was created. The rules of 8.5-ball are identical to those of 8-ball, except that the player may only shoot at the lowest numbered object ball remaining in their set. This constrains the shots available, and the positional options available, in a manner similar to 9-ball. The search depth experiments for the Probabilistic and Monte-Carlo search algorithms were repeated for this game under the E_{Taiwan} error model. The results of these matches are shown in Table 4.12 and Table 4.13.

Now in 8.5-ball using Probabilistic search, depth 2 search won 56-44 games against depth 1. While this is still an inconclusive match result, it is very strong evidence of an improvement over the previous 34-66 loss suffered by depth 2 search in 8-ball under the same error model. 3-ply and 4-ply search still fare worse than 1-ply. The 4-ply match result is close enough to be inconclusive, but considering the 3-ply result, it is highly unlikely that 4-ply search is better than 1-ply.

Using Monte-Carlo search in 8.5-ball, 2-ply search clearly and statistically significantly outperforms 1-ply search, winning 60-40. Like with Probabilistic search, this is a substantial improvement from the 8-ball result of 45-55 under the same error model. Overall, these results add up to the properties of this game favouring deeper search, much more so than then game of 8-ball. This is an interesting result, as it shows that the importance of search

Match	W	SIS	SS	Bank	Kick	Combo
No Banks	55	512/631=81.1%	15/43	N/A	N/A	N/A
Banks	45	508/633=80.3%	9/29	5/11	2/9	4/4
8.5-Ball No Banks	47	420/638=65.8%	131/240	N/A	N/A	N/A
8.5-Ball Banks	53	430/678=63.4%	100/166	1/10	1/25	5/10

Table 4.14: Bank/Kick/Combo match result

depth in billiards games, as well as being a function of the error model, is a function of the properties of the particular game being played.

Note also the higher proportion and success rate of safety shots in 8.5-ball, under both algorithms. The greater proportion of safety shots is a result of the player being required to target one specific ball. More often there will be no shot available on that ball, so the player will have to perform a safety. The higher success rate also follows. Since it is known which ball the opponent will have to target, it is easier to find a safety shot that prevents that player from having a shot on that specific ball.

4.3.8 Bank, Kick, Combination shots

To determine the benefit of bank, kick, and combination shots, a variant with these shot types enabled and activated at $b_0 = 0.8$ was played against a variant that never attempted these shots. The results are shown in Table 4.14. The Bank column lists the ratio of successful to attempted bank shots. The Kick column lists the ratio of successful to attempted kick shots. The Combo column lists the ratio of successful to attempted combination shots. The same experiment was repeated for 8.5-ball, with the results shown in the same table.

In 8-ball, the total number of banks attempts constituted a small percentage of the total overall shots. 24 out of 686 shots by the banking variant were bank, kick, or combination attempts. The match result was too close to call. Since such a small proportion of total shots were bank, kick, or combo attempts, the impact of these shots on the final result was likely negligible. Combination shots were the least frequent, but most successful. Bank shots were the most frequent and somewhat successful. Kick shots were almost as frequent as bank shots, but substantially less successful. This is as expected. Bank shots are more robust than kick shots, as the angle at which an object ball rebounds off a rail is less sensitive to initial shot parameters than the angle for a cue ball. Opportunities for good

Match	W	Average Tree Size	Average Time (s)
Depth 1, No Pruning	45	8.07	2.75
Depth 1, Pruning	55	5.27	3.14
Depth 2, No Pruning	50	54.48	3.19
Depth 2, Pruning	50	18.08	2.42

Table 4.15: Effect of pruning in Probabilistic search

Match	W	Average Tree Size	Average Time (s)
Depth 1, No Pruning	45	74.79	2.14
Depth 1, Pruning	55	58.66	1.50
Depth 2, No Pruning	48	4239.76	70.49
Depth 2, Pruning	52	2786.01	45.41

Table 4.16: Effect of pruning in Monte-Carlo search

combination shots are rare, as object balls must be lined up to make them feasible, *and* the player must only have a clear shot at the kick shot; if any straight-in shots are available, they will almost certainly be a better option.

The experiment was repeated for 8.5-ball to see the impact of these shots in this game. It seemed plausible that a banking player might have an advantage in this game, as they would be able to pocket their one target ball even when no straight-in shots at it were available. In practice, this was not the case. Many such attempts were made, but they almost all missed. It turned out that, in general, finding a bank, kick, or combo shot on a specific ball is very difficult under then E_{Taiwan} error model. The banking variant in 8-ball had the advantage that it could generate bank shots on all balls and choose the best one. There was more likely to be a good shot available amongst this larger set.

Note that the banking player executed fewer safeties than the non-banking player in both cases. Sometimes there would be bank, kick, or combo shots available to attempt in states where no straight-in shots were available. The non-banking player would always take a safety shot in this situation, so took more safeties overall.

4.3.9 Search Enhancements

Table 4.15 shows the effect of pruning in Probabilistic search. Matches were played between programs with pruning enabled and disabled, to 1-ply and 2-ply depth. Variants with

and without pruning should be logically identical, so should have a 50-50 underlying win rate. The wins for each side are listed as a sanity check. The average tree size (in nodes) and average search time (in seconds) are listed for each variant. All of a program's regular searches over the 100-game match are included in these averages. Safety searches are not included in these averages, as the method is the same for all variants. Table 4.16 shows the result of the same experiment repeated for the Monte-Carlo search algorithm.

Pruning clearly reduces the overall tree size substantially, in all cases. The match results were all within expected bounds for a 50-50 win rate, so the sanity check passed. The effect on search time for all variants except 2-ply search was not substantial, because so few total nodes were searched in these cases. Search times were measured on the server side, and include round-trip times for the messages over the internet, so the differences in average times in these cases are as likely to be the results of variance in packet round-trip times as they are to be a direct result of pruning. For 2-ply Monte-Carlo search, however, the benefit in both nodes searched and average search time is clear and significant. The decrease in average search time is roughly proportional to the decrease in average tree size, as would be expected. Pruning is an effective optimization.

4.3.10 Search Algorithm Comparison

Experiments were constructed to compare the main search algorithms used by PickPocket. A tournament was played between the following versions of the program:

- Greedy: This baseline algorithm runs the shot generator for the table state, and executes the shot with the highest probability estimate. No search is performed. Greedy algorithms were used to select shots in [5, 6, 7, 8].
- Prob: The Probabilistic search algorithm.
- MC: The Monte-Carlo search algorithm with $num\_samples = 15$.

Matches were played between each pair of algorithms, under the E_{low}, E_{Taiwan}, and E_{high} error models described earlier. These model a professional, strong amateur, and beginning player, respectively. Under each error model, each of the search algorithms was run to the best performing search depth for that amount of error, as determined in

Match	W	SIS	SS
	E_{low}		
Greedy	7	202/377=53.6%	43/68
Prob	93	659/726=90.8%	74/128
Greedy	3	122/249=49.0%	23/38
MC	97	662/722=91.7%	29/59
Prob	38	375/432=86.8%	15/40
MC	62	510/566=90.1%	12/28
	E_{Taiwan}		
Greedy	19	411/745=55.2%	74/119
Prob	81	631/829=76.1%	138/222
Greedy	22	341/604=56.5%	35/81
MC	78	637/806=79.0%	66/135
Prob	44	525/675=77.8%	74/145
MC	56	552/730=75.6%	63/109
	E_{high}		
Greedy	27	381/1136=32.4%	79/119
Prob	73	579/1145=50.6%	192/273
Greedy	25	368/1039=35.4%	59/89
MC	75	554/1023=54.2%	101/153
Prob	33	432/972=44.4%	98/155
MC	67	495/1040=47.6%	77/131

Table 4.17: Comparison of search algorithms

Sections 4.3.5 and 4.3.6. Thus, under E_{low}, both algorithms searched to 2-ply depth. Under E_{Taiwan}, both algorithms searched to 1-ply depth. Under E_{high}, Monte-Carlo searched to 2-ply depth and Probabilistic to 1-ply depth. Table 4.17 shows the tournament results.

Both search algorithms defeated Greedy convincingly under all error conditions. This demonstrates the value of lookahead in billiards. Greedy selects the easiest shot in a state, without regard for the resulting table position after the shot. The search algorithms balance ease of execution of the current shot with potential for future shots. Thus, they are more likely to have easy follow up shots. This wins games.

Under each error model, the algorithms vary in their percentage of completed straight-in attempts. This highlights the differences in position play strength between the algorithms. Since the same error model applies to all algorithms, they would all have the same straight-in completion percentage if they were seeing table states of equal average quality.

Lower completion rates correspond to weaker position play, which leaves the algorithm in states that have a more challenging 'best' shot on average. Completion rates generally increased from Greedy to Probabilistic to Monte-Carlo search, with the difference between Greedy and Probabilistic being much greater than that between Probabilistic and Monte-Carlo search.

Under E_{high}, the games tended to be longer, as the lower accuracy led to more missed shots. Under E_{low}, matches completed faster with fewer misses. The change in straight-in completion rate for a given algorithm between error models represents this change in accuracy. Winning programs take more shots than losing programs, as they pocket balls in longer consecutive sequences.

In 8-ball, since a player may aim at any of his assigned solids or stripes, there are usually straight-in shots available. Safeties, attempted when no straight-in shots could be generated, totalled roughly 10% of all shots in the tournament. Therefore at least one straight-in shot was found in 90% of positions encountered. This demonstrates the rarity of opportunities for bank, kick, and combination shots in practice, as they would be generated only when no straight-in shots are available. Even then, safety shots would often be chosen as a better option. Safeties were more effective under E_{high}, frequently returning the turn to the player. They were generally less effective under E_{low}, as the increased shot accuracy led to there being fewer states from which the opponent had no good straight-in shots available.

Monte-Carlo search is clearly the strongest of the algorithms. Under all error models, it defeated Greedy by a wide margin, and then defeated Probabilistic search in turn. The victories over Probabilistic search under E_{low} and E_{high} have statistical confidence, while the E_{Taiwan} result is too close to call. Overall, this suggests that the value of sampling and taking into account the range of possible shot outcomes is substantial under a wide range of error models. This is in agreement with the results of the previous experiments on search depth and search breadth.

4.4 Computer Olympiad 10

PickPocket won the first international computational 8-ball tournament at the 10^{th} Computer Olympiad [25]. Games were run over a *poolfiz* server, using the E_{Taiwan} error model

Rank	Program	1	2	3	4	Total Score
1	PickPocket	-	64	67	69	200
2	PoolMaster	49	-	72	65	186
3	Elix	53	54	-	71	178
4	SkyNet	53	65	55	-	173

Table 4.18: Computer Olympiad 10 competition results

detailed earlier. PickPocket used the Monte-Carlo search algorithm for this tournament, searching to 2-ply depth.

The tournament was held in a round-robin format, each pair of programs playing an eight game match. Ten points were awarded for each game won, with the losing program receiving points equal to the number of its assigned solids or stripes it successfully pocketed. PickPocket scored more points than its opponent in all three of its matches. The results of the tournament are shown in Table 4.18.

4.5 Computer Olympiad 11

PickPocket also won the 11^{th} Computer Olympiad, held in Italy in May 2006. The same setup was used, with a slightly different error model: E_{Italy} has parameters {0.125, 0.1, 0.075, 0.5, 0.5}. This corresponds to a somewhat more accurate player. Tests showed than under E_{Italy}, 75.66% of straight-in shots in random table states were successful, over a 10,000 shot sample size. Under E_{Taiwan}, the same test resulted in 69.07% of shots being pocketed.

One shortcoming of the 10^{th} Olympiad was that programs only played 8-game matches against one another. While the results certainly suggested that PickPocket was the top program, there was no way of knowing whether this was really the case or if PickPocket won because of fortuitous random events. Counting balls pocketed in a loss helped overcome the effects of variance somewhat, as a stronger program is likely to pocket more balls when it loses. However, the overall results cannot be claimed to be statistically significant (comparing PickPocket with the second place program, $\underline{t}(471) = 1.62\ \underline{p} = 0.11$).

To address this, the 11^{th} Olympiad featured a 50-game match between each pair of competing programs. Because this is a larger sample size, balls pocketed in a loss were not

Rank	Program	1	2	3	4	5	Total	Win %age
1	PickPocket	-	38	34	37	39	148	74.0%
2	Elix	12	-	26	31	36	105	52.5%
3	SkyNet	16	24	-	28	37	105	52.5%
4	PoolMaster	13	19	22	-	22	76	38.0%
5	Snooze	11	14	13	28	-	66	33.0%

Table 4.19: Computer Olympiad 11 competition results

counted. The scores represent the number of games out of the 50-game match won by each player. The results of the tournaments are shown in Table 4.19.

Again PickPocket scored more wins than its opponent in all of its matches, winning 68-78% of games versus all opponents. Overall it won 74% of the games it played, defeating the next best performers by a substantial margin. This is a convincing, statistically significant result (comparing PickPocket with the second place programs, $\underline{t}(391) = 4.58\,\underline{p} = 0.0$). The version of PickPocket that competed in this tournament was substantially tweaked and improved from the previous year, and had enabled all of the features described in this document.

A variety of approaches were used by the other entrants in the tournament, which were developed concurrently with PickPocket. The main distinguishing features of each program are as follows:

Elix

Elix was developed by a strong billiards player who translated his knowledge of the game into an ad-hoc, rule based approach to shot selection. A set of shots are generated, and then a sequence of rules is used to select which shot to execute. These rules operate on such inputs as the shot difficulty parameters, the results of a 1-ply Monte-Carlo type search, and other features of the table state. Because of complex interactions between rules, Elix would sometimes make poor shot selections. Contrast this with the uniform approach used by PickPocket, where every shot (save safeties) is found as a result of the same search process. Since PickPocket has few special cases, there is less risk of them being inadvertently activated.

SkyNet

Leckie's SkyNet [24] was the program most similar to PickPocket, using a primarily search-based approach. The search algorithm used was based on Expectimax, and was similar to the Probabilistic Search algorithm detailed for PickPocket. SkyNet searched to a 3-ply depth. To estimate the probability of success of a shot, a runtime Monte-Carlo sampling algorithm was used. This formed the multiplier for each child node's returned value. Contrast this with PickPocket's more efficient, but less accurate, use of a lookup table to estimate each shot's probability of success. SkyNet's search, like Probabilistic search, performed a deterministic expansion of nodes. Unlike Monte-Carlo search, it did not account for the range of possible shot outcomes. SkyNet's leaf evaluations were based on the number of shots available at leaf nodes, whereas PickPocket's evaluation is based on the number and quality of those shots.

PoolMaster

Leckie describes two paradigms for billiards shot generation in [24]: shot discovery and shot specification. PickPocket, Elix, and SkyNet all use shot discovery methods. Here, shots leading to position play are generated by blindly varying an initial shot to create a set of shots which will all lead to different final cue-ball positions. Search then 'discovers' the shots among these that lead to good position play. In contrast, a shot specification approach to generation explicitly chooses good final cue ball positions, and then uses sampling and optimization to find shots that leave the cue ball as near as possible to these positions. Shot discovery is computationally cheap, as no physics simulation is required. Shot specification is expensive, as extensive physics simulations are required to find a shot that best leaves the cue ball in the desired final position.

PoolMaster [26] uses a shot specification approach to move generation; positions on the table are scored to find good positions for the cue ball to come to rest after the shot. Local maxima are used as candidates, similar to PickPocket's ball-in-hand play. The optimization algorithm used to direct sampling to find shots that leave the cue ball in these positions is described in [27]. PoolMaster performs 1-ply search on the generated shots. However, it used a basic search algorithm which did not take into account the probability of success of the generated shots. PoolMaster therefore often made risky shots in tournament play,

which likely would not have been chosen if their difficulty had been taken into account.

Snooze

Snooze was a late entrant to the tournament, and the version that competed was still a work-in-progress. No details of its operation are available.

Chapter 5

Conclusions and Future Work

5.1 Future Directions for Work

5.1.1 Adaptation to Other Billiards Games

The major differences between billiards games fall into two categories: ball ownership and sinking order. Ball ownership can be individual or communal. Under individual ownership, each player is assigned a set of balls which are 'theirs', to either sink or protect. Under communal ownership, the object balls are not specifically assigned to individual players. Sinking can be ordered or unordered. Players are either required to sink the object balls in a specific order, or are free to shoot at any (or a subset of) the object balls.

8-ball features individual ball ownership, as each player is either solids or stripes. Sinking is unordered, as players may shoot at any object ball in their set.

Several other billiards games can be summarized as follows:

9-ball

9-ball is played with 9 numbered object balls. Ball ownership is communal. The object is to be the first player to legally pocket the 9-ball. Players must always shoot at the lowest numbered ball remaining on the table, so sinking is ordered.

Snooker

Snooker is played on a large table with 21 object balls, divided into a set of 15 reds and a set of 6 colours. Ball ownership is communal. The game is played in two phases. In the first phase, players alternate between aiming at reds and colours, switching back and forth after

every successful shot. In this phase sinking is unordered, as players may aim at any ball in the designated set. Coloured balls pocketed in this phase are replaced on the table. The second phase begins once all of the reds have been pocketed. This phase features ordered sinking, as players must pocket the colours in a specific sequence.

One Pocket

One pocket is played with 15 object balls. Each player is assigned one of the two pockets at the foot end of the table, and scores points by pocketing the communal balls in this pocket. The object balls are communal and may be pocketed in any order.

Cutthroat

Cutthroat is a game for 3 or 5 players. 15 numbered object balls are divided into equally sized sets and assigned to the players. Players try to pocket their opponents' balls and protect the balls in their own set. Once a player's entire set has been pocketed, they are eliminated from the game. Opponent modelling may be useful in cutthroat, as a common human strategy is to try to eliminate the stronger players first, leaving the player only facing weaker opponents towards the end of the game.

The overall search approach used by PickPocket can easily be applied to any of these games, by modifying the move generator to only generate legal shots according to the rules of the game, and modifying the shot result evaluator to follow the rules of the new game. To obtain the best performance, the breakdown of where PickPocket spends its time when finding a shot may need to be modified. Given a fixed amount of time to search, how should that time be invested? A broad shallow search could be performed, or a narrower deeper search. Safeties could be considered more frequently, or less frequently, and the amount of time spent generating safeties when they are considered could be varied. How to optimally distribute time amongst these tasks depends on the properties of the game being played.

The experiments with 8.5-ball showed that deeper search has a larger benefit when object balls must be pocketed in a specific order. When the player may shoot at balls in any order, as in 8-ball, then sequences of balls can be strung together by looking 1-ply ahead. This is sufficient to find a shot that is likely to sink the targeted object ball, plus leave

position on a follow-up ball. Since there are many potential target balls for the current shot, and many potential follow-up balls for the next shot, it is likely that a good shot can be found from a wide range of table states. Deeper search confers marginal additional benefit. When object balls must be pocketed in a specific sequence, on the other hand, there is only one possible target for the current shot, and only one ball to leave position on for the next shot. Now from many table states no good shots may be physically possible that both pocket the desired object ball and leave position on the follow-up ball. Deeper search makes it less probable that a program will end up in such a state.

Deeper search, therefore, seems most applicable to games such as 9-ball, and the second phase of snooker, where object balls must be pocketed in a specific sequence. 1-ply search may be sufficient for games where players have more options for what to shoot at, and a program for these games may best invest its time in a broader search, considering more potential shots.

Games with communal ball ownership emphasize safety play more so than games with individual ownership. This is because a sequence of successful shots followed by a miss is very bad for a player. Each successful shot cleared a ball from the table, making the task for their opponent easier when it switches to their turn. If a miss after several shots is deemed likely, it may be better to play a safety and leave the opponent with the most challenging possible table state, rather than a partially cleared table. Professional 9-ball matches often feature players trading safeties back and forth, until one player breaks out and runs the table. Strategic safety play is similarly important in snooker - the term 'snooker' means to leave one's opponent without a clear shot.

Similarly, safeties are most effective when balls must be pocketed in a specific order. It is much easier to leave a player with no good shots on a specific ball, than to leave them with no good shots on any ball in a set. This can also be seen from the experiments with 8.5-ball. Therefore, safeties should be considered more frequently in games that have ordered pocketing and communal ball ownership, than in games featuring unordered pocketing and individual ownership.

5.1.2 Implementation Enhancements

There are a number of ways in which PickPocket's performance could potentially be enhanced. This section describes several such ideas.

Parameter Optimization

PickPocket has many tunable parameters, including the number of shot variants to generate per ball/pocket combination, the evaluation function parameters, and the thresholds used for safety and bank shot activation. These have been hand-tuned to values that are reasonable, but are not necessarily optimal. To precisely hand-tune these parameters would be an overwhelming task, as a large sample size of games is required to evaluate the impact of any changes. A machine learning algorithm could be applied to automate the tuning of these parameters over the course of thousands of games. While the experimental results from Sections 4.3.2, 4.3.3, and 4.3.8 suggest that the benefit to precisely tuning such parameters individually is marginal, a program variant with all of its parameters tuned may have a substantial edge over the current version.

Evaluation Function Terms

PickPocket's current evaluation is based on the number and quality of shots available in leaf node states. There are several other examinable features of a table state which could give an indication of how strong it is for the player. Two such possibilities are the *centredness* and *clusteredness* of the state.

Centredness refers how centred a player's balls are compared to their opponent's. A ball towards the centre of the table is pocketable from all sides; no matter where the cue ball is, there are likely several good options for sinking it in a range of pockets. Conversely, a ball along a rail may only be easily pocketable from one side, into one specific pocket. The more a player's balls tend towards the centre of the table, and the more their opponent's balls tend towards the rails, the better that state should be for the player.

Clusteredness refers to how clustered a player's balls are compared to their opponent's. Tightly clustered object balls are hard to pocket, as they interfere with each other's lanes to the pockets. Unclustered object balls are easier to pocket, as they are more likely to have a clear path to a wider range of pockets, and more likely to have a clear path for the cue ball

to approach along. The more a player's balls are unclustered, and the more their opponent's are clustered, the better that state should be for the player.

Both of these terms could be numerically evaluated and incorporated into a complex evaluation function. It is not obvious how the terms should be relatively weighted - extensive tuning, perhaps via machine learning, would be required to find reasonable weights.

Progressive Pruning

The Monte-Carlo search algorithm could benefit from progressive pruning, as described in [19]. This technique finds shots to prune that are statistically unlikely to exceed the current best score based on the samples already seen. Consider a case where the values 0.35, 0.41, 0.22, and 0.27 are seen in the first 4 of 6 samples, and average values of 0.85 or higher are needed on the remaining 2 samples in order to exceed the current best score. Based on the samples seen so far, it seems unlikely that the next samples would have values high enough to give this shot the best score. This is a case where progressive pruning may find a cutoff. Note that the pruning method shown in Section 3.3.3 would not find a cutoff because it is still mathematically possible for the coming samples to have values that would give this shot the best score. Unlike this method, progressive pruning may in extreme cases alter the search result by pruning what would be found as the best branch by a complete search.

Progressive pruning is less applicable to billiards than games such as Go, however, because of the relatively small number of samples taken per shot. With $num\_samples = 15$, the scores found are not going to be highly accurate, as even one outlier can have a significant impact on the average score. The running average after 5 or 10 samples will be even less accurate. These could still be used to eliminate branches that are statistically unlikely to improve. However, due to the lack of resolution of the success probability averages, either a conservative threshold could be set and only cases highly unlikely to improve would be pruned (but potentially missing opportunities for pruning), or an aggressive threshold could be set that finds many cutoffs (but with a high risk of pruning too many shots and altering the search result). The larger the sample size, the better a balance between these extremes can be struck.

Adversarial Search

Billiards differs from many traditional games in that the sequence of play is not strictly ordered. Players continue shooting until they fail to pocket a ball, rather than trading the turn back and forth every shot. PickPocket's search algorithms only consider the shots available to itself; it does not continue searching after a miss and the turn passes to the other player. Therefore the quality for the opponent of the resulting positions after probable misses is not taken into account in the best shot found by the search. For Probabilisitic search, this would be difficult to incorporate, as generating 'average' probable misses is not straightforward. However, it would be easy to add adversarial search to the Monte-Carlo search algorithm. Instead of stopping after a miss, the resulting table state could be searched from the opponent's perspective, and the negative of the node score propagated up the tree. This may improve performance, leaving the opponent in worse on average positions after a miss. On the other hand, the time spent searching opponent's positions may be better invested searching more shot options for the player. Experimentation would be required to determine whether there is any benefit to performing adversarial search in billiards.

Ball-in-hand Play

PickPocket's approach to ball-in-hand situations efficiently generates a range of positions from which it is known that there is at least one good shot. However, it does not take into account position play - it is left up to the search algorithm to select the position and shot that leads to the best prospects further ahead in the game. Human players know to take advantage of ball-in-hand to pocket problem balls that may otherwise be difficult to gain good position on. Balls near to the rails, and in the vicinity of other object balls, are most likely to fall into this category. Therefore, an improved ball-in-hand algorithm might bias the candidate positions generated towards those with good shots on such balls, and away from those object balls that are unclustered and towards the centre of the table. This would likely lead to an improvement in the average position seen later in the game.

Parallelization

PickPocket's search algorithms are easily parallelizable. Each shot simulation is an independent unit of execution, and since these simulations are where PickPocket spends a majority of its time, substantial speedup could be gained by distributing this work between a number of processors. For Monte-Carlo search especially, the samples taken for each shot at a node could be split up amongst processors, allowing a greater number of total samples to be taken. Care must be taken when designing a parallel algorithm to ensure that cutoffs are found efficiently - this is one of the biggest challenges when parallelizing search algorithms. Communication is required to ensure that each processor has the most current cutoff threshold. If cutoffs are ignored, then every branch of a search tree could be searched independently with no communication requirements. Parallelization may also allow Monte-Carlo search to reach deeper search depths in a reasonable amount of time. This may be of greater benefit to games such as 9-ball than 8-ball.

Learning an Error Model

Given a table with an unknown error model, take N shots and build an error model. This feature would be useful for automatically calibrating a robot to a physical table. The error model could be further refined on the fly based on the shot results seen during gameplay.

5.1.3 Man-Machine Challenge

A major goal of any game-playing project is to be able to defeat top human players. This has been done in chess and checkers, and the Robocup project aims to defeat a top human team at soccer by the year 2050. Defeating top humans is a strong demonstration of effectiveness, and a goal that motivates the development of new techniques. Ultimately billiards robots will be strong enough to accomplish this, and may even become common as practice opponents, just as chess players now play against computer programs for practice and entertainment.

For a billiards robot to challenge a top human, three components must be in place:

1. The robot must have shot error margins roughly as low as the top humans. The exact accuracy required to win depends on how the AI's shot selection compares to the

top human's shot selection. If the AI's shot selection is weaker than the human's, the robot must be more accurate than the human competitor. If it has better shot selection than the human, then the robot may not need to be quite as accurate as the human to win the match.

2. Physics simulation must be accurately calibrated to the physical table. The AI's shot selection is based on the results of physics simulation. If the simulation accurately predicts what will happen on a physical table as a result of a shot, then the shots found by the AI will be effective.

3. The shot selection of the AI driving the robot must be roughly as good as the top human. Again, the exact requirements depend on the physical accuracy of the robot being driven.

Early forms of all three components are now in place. Soon full games will be held between the robot Deep Green [1] and human challengers. The physical accuracy of the robot, and calibration of the physics simulation, are still a long way from being sufficient to challenge the best humans. The current systems should provide an entertaining challenge for humans, and should have a chance of defeating casual players.

It is unclear how the current version of PickPocket would fare against a top human player, as there is no way to directly compare just shot selection. Without a robot that is similar in accuracy to strong humans, it would be difficult to tell (except for obvious mistakes) whether losses by the robot were due to weak shot selection, or a lack of sufficient physical accuracy. Having an expert human comment on PickPocket's shot selection may provide insight on the strength of its strategic play. However in some games, such as backgammon, non-conventional moves found by computers actually turned out to be better than the established wisdom, and resulted in changes over time to the strategies employed by humans.

The properties of the game chosen for a man-machine challenge may have an impact on the robot's chance of success. In 8-ball, 1-ply search is sufficient to consistently find good shots; depending on the error model, there may be little advantage to additional lookahead. A robotic player may therefore have a better chance of defeating a top human at 8-ball than

9-ball, where lookahead is more important, and the best human players can plan a path to clear the entire table.

5.2 Conclusions

This thesis described PickPocket, an adaption of game search techniques to the continuous, stochastic domain of billiards. Its approach to move generation, evaluation function, and the Probabilistic and Monte-Carlo search algorithms were described. Pruning optimizations for both search algorithms were detailed. The technique of estimating shot difficulties via a lookup table was introduced. A range of other techniques that PickPocket uses for efficiency and effectiveness were described.

Experimental results proved the benefit of lookahead search over the previously standard greedy technique. They demonstrated that Monte-Carlo search is the strongest of the two presented search algorithms under a wide range of error conditions. They suggest that search breadth is more important than search depth in the game of 8-ball. Additionally, they suggest that the exact parameters used in the evaluation function, to control the Monte-Carlo search algorithm, and for bank, kick and combination shot activation are not of major consequence. They show that, in 8-ball, playing too many safety shots is a poor strategy.

The exact benefit of search depth and safety shots was shown to depend on both the amount of noise added to each shot, and the properties of the specific billiards game being played. This suggests that these factors should be kept in mind when adapting to various robotic platforms and alternate billiards games. This should prove useful as the next Computer Olympiad competition will feature a billiards game other than 8-ball.

PickPocket proved itself the world's best billiards AI at the 10^{th} and 11^{th} Computer Olympiad competitions.

A man-machine competition between a human player and a billiards robot will soon occur. This research goes a long way towards building an AI capable of competing strategically with strong human players.

Bibliography

[1] Fei Long, Johan Herland, Marie-Christine Tessier, Darryl Naulls, Andrew Roth, Gerhard Roth, and Michael Greenspan. Robotic pool: An experiment in automatic potting. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 04)*, volume 3, pages 2520–2525, 2004.

[2] Mohammad Ebne Alian, Saeed Bagheri Shouraki, M.T. Manzuri Shalmani, Pooya Karimian, and Payam Sabzmeydani. Roboshark: A gantry pool playing robot. In *35th International Symposium on Robotics (ISR 2004)*, 2004. Electronic Publication.

[3] Bo-Ru Cheng, Je-Ting Li, and Jr-Syu Yang. Design of the neural-fuzzy compensator for a billiard robot. In *Networking, Sensing, and Control*, volume 2, pages 909–913, 2004.

[4] Sang William Shu. *Automating Skills Using a Robot Snooker Player*. PhD thesis, Bristol University, 1994.

[5] Mohammad Ebne Alian and Saeed Bagheri Shouraki. A fuzzy pool player robot with learning ability. *WSEAS Transactions on Electronics, Issue 2*, 1:422–426, Apr. 2004.

[6] Mohammad Ebne Alian, Saeed Bagheri Shouraki, and Caro Lucas. Evolving strategies for a pool player robot. *WSEAS Transactions on Information Science and Applications, Issue 5*, 1:1435–1440, Nov. 2004.

[7] S.C. Chua, W.C. Tan, E.K. Wong, and V.C. Koo. Decision algorithm for pool using fuzzy system. In *Artificial Intelligence in Engineering & Technology*, pages 370–375, June 2002.

[8] Z.M. Lin, J.S. Yang, and C.Y. Yang. Grey decision-making for a billiard robot. In *Systems, Man, and Cybernetics*, volume 6, pages 5350–5355, Oct. 2004.

[9] Tony Jebara, Cyrus Eyster, Josh Weaver, Thad Starner, and Alex Pentland. Stochasticks: Augmenting the billiards experience with probabilistic vision and wearable computers. In *ISWC '97: Proceedings of the 1st IEEE International Symposium on Wearable Computers*, page 138, Washington, DC, USA, 1997. IEEE Computer Society.

[10] Lars Bo Larsen, Morten Damm Jensen, and Wisdom Kobby Vodzi. Multi modal user interaction in an automatic pool trainer. In *International Conference on Multimodal Interfaces (ICMI 02)*, pages 361–366, 2002.

[11] Murray Campbell, A. Joseph Hoane Jr., and Feng-Hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

[12] Jonathan Schaeffer, Robert Lake, Paul Lu, and Martin Bryant. Chinook: The world man-machine checkers champion. *AI Magazine*, 17(1):21–29, 1996.

[13] Donald Knuth and Ronald Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[14] Bruce W. Ballard. The *-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, 21(3):327–350, 1983.

[15] Thomas Hauk. Search in Trees with Chance Nodes. Master's thesis, University of Alberta, January 2004.

[16] Matthew L. Ginsberg. Gib: Steps toward an expert-level bridge-playing program. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 584–593, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[17] Darse Billings, Aaron Davidson, Jonathan Schaeffer, and Duane Szafron. The challenge of poker. *Artificial Intelligence*, 134(1-2):201–240, 2002.

[18] Nathan Reed Sturtevant. *Multiplayer games: algorithms and approaches*. PhD thesis, UCLA, 2003. Chair-Richard E. Korf.

[19] Bruno Bouzy and Bernard Helmstetter. *Advances in Computer Games, Many Games, Many Challenges*, chapter Monte Carlo Go developments, pages 159–174. 2003.

[20] Brian Sheppard. World-championship-caliber scrabble. *Artificial Intelligence*, 134(1-2):241–275, 2002.

[21] Michael Chung, Michael Buro, and Jonathan Schaeffer. Monte carlo search for real-time strategy games. In *IEEE Symposium on Computational Intelligence and Games*, pages 117–124, 2005.

[22] Will Leckie and Michael Greenspan. An event-based pool physics simulator. In *Proc. of Advances in Computer Games 11*, Sept 2005. To appear.

[23] Billiards Congress of America, editor. *Billiards: The Official Rules and Records Book*. The Lyons Press, 2002.

[24] Will Leckie and Michael Greenspan. Monte carlo methods in pool strategy game trees. In *Proc. of Computers and Games 2006*, May 2006. To appear.

[25] Michael Greenspan. UofA Wins the Pool Tournament. *International Computer Games Association Journal*, 28(3):191–193, Sept. 2005.

[26] Jean-Pierre Dussault and Jean-Francois Landry. Optimization of a billiard player - tactical play. In *Proc. of Computers and Games 2006*, May 2006. To appear.

[27] Jean-Pierre Dussault and Jean-Francois Landry. Optimization of a billiard player - position play. In *Proc. of Advances in Computer Games 11*, Sept 2005. To appear.